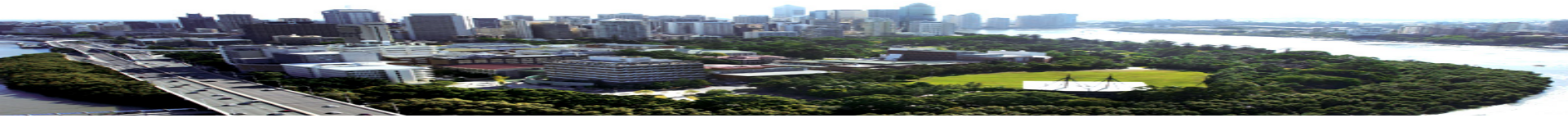
ENB 350 Real-time Computer based systems

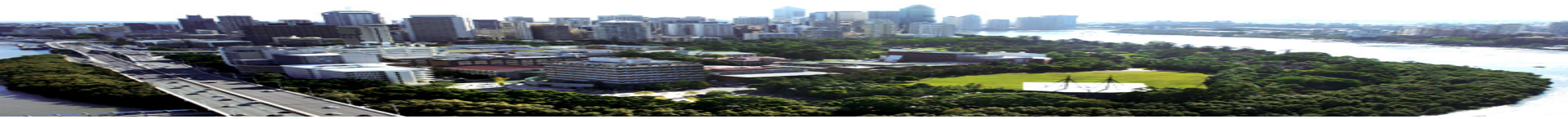
Lecture 8 –PREEMPTIVE MULTITASKING

V. Chandran



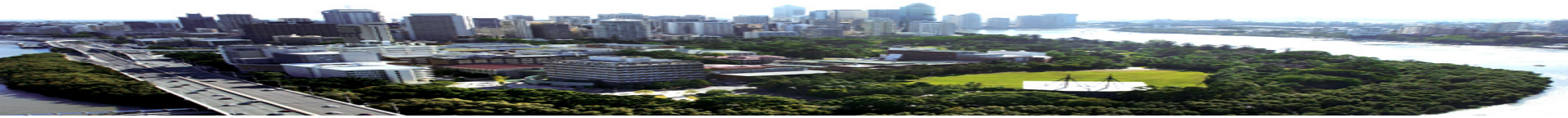
Contents

- Preemptive multitasking
- MicroC/OS-II



Pre-emptive multitasking

- Context switching is triggered when interrupts are serviced. This could be any interrupt
- Every thread may also be given a maximum time slice. Then a timer interrupt triggers the context switch.
- System response time is improved greatly and worst case response times can be guaranteed



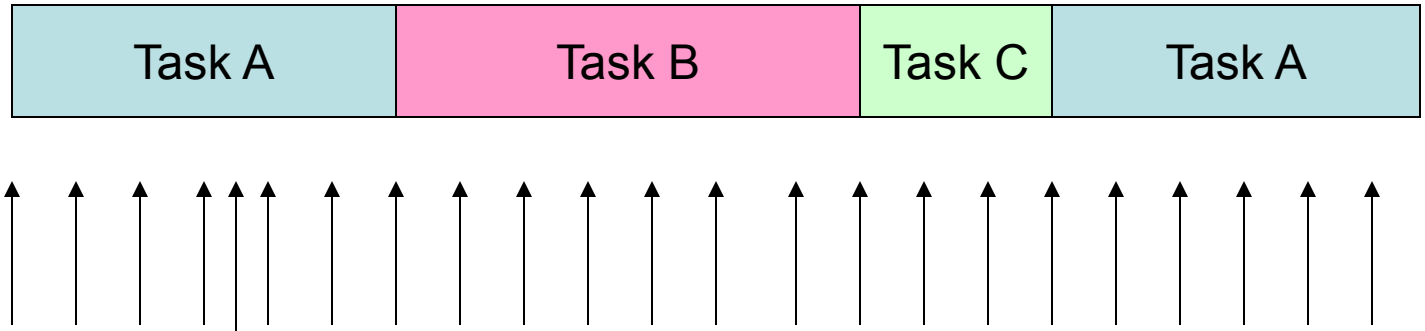
Preemptive - types

- In immediately pre-emptive multitasking the scheduler is invoked at the end of every interrupt service and a context switch may take place
- Dynamic C slice is not immediately pre-emptive. ISRs return to the same task. Task switch only takes place after time slices allocated to the task are used. In other systems, a switch can take place at the next timer interrupt or tick.
- Micro C OS-II is a real-time kernel (developed by Jean Labrosse) that has been ported to the R4000 also and can be included as a library. It is immediately pre-emptive.

Preemptive -Time Slice

Response time = can be very long

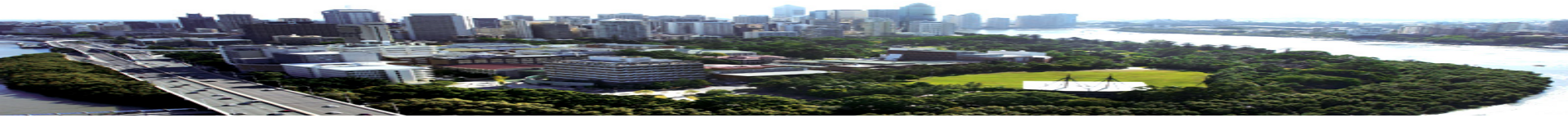
The remaining slice for A if Task B were the next ready task.



Timer interrupts or ticks

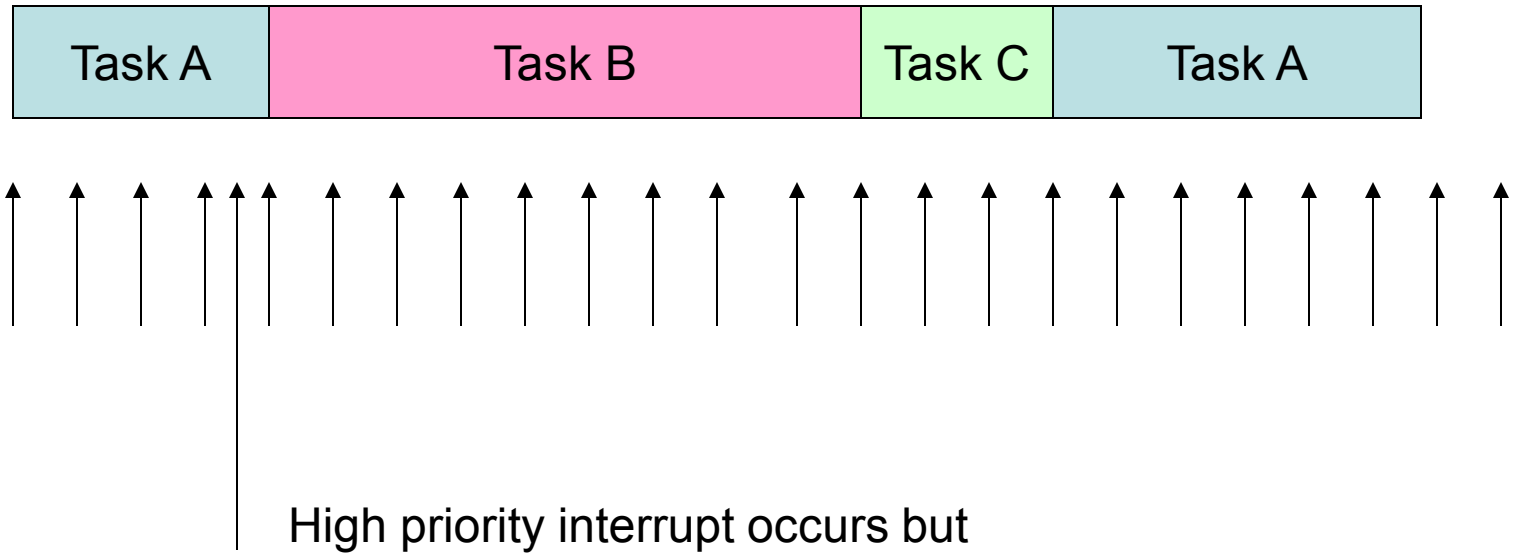
Task A runs for specified number of ticks

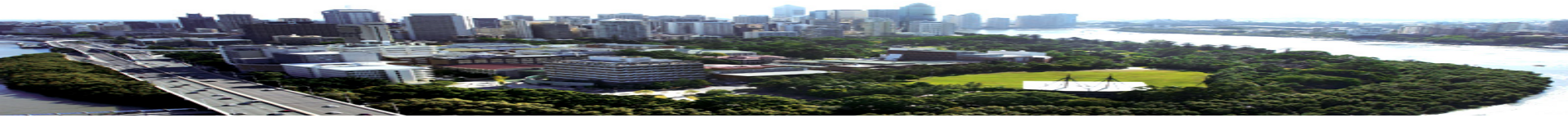
High priority interrupt should be serviced
by Task B but has to wait



Preemptive not immediate

Response time = 1 tick in the worst case if tasks are prioritized

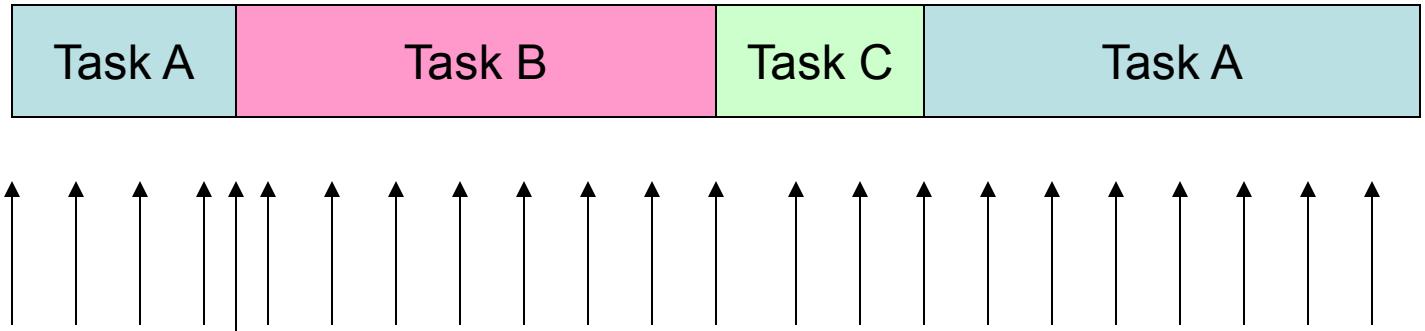




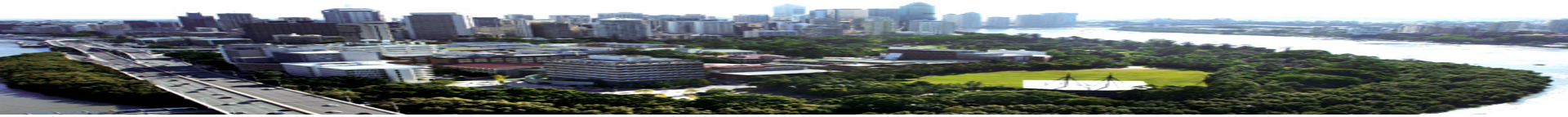
Immediate preemptive

Response time = interrupt latency + scheduling latency

Assuming task B is the highest priority task

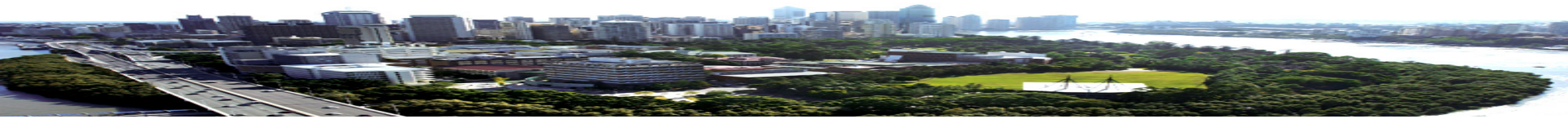


High priority interrupt – Task A is suspended and Task B is made to run to process some critical event. There is a small delay arising from interrupt processing and context switch



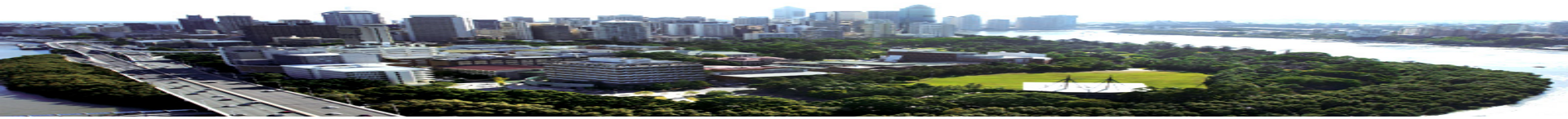
Task Priorities

- Preemptive multitasking associates a priority level with each task.
- RTOS allows a programmer to set these. They can also modify priorities dynamically.
- If many tasks are ready to run, the CPU should always run the highest priority task



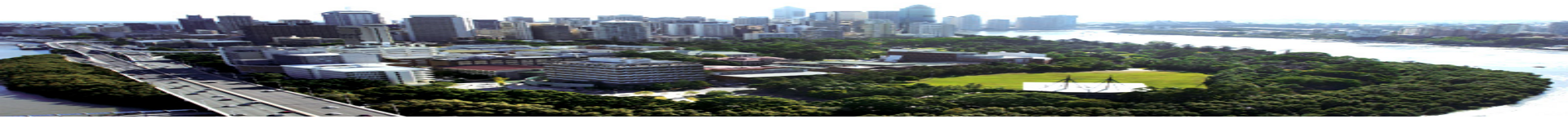
What is required?

- A real-time kernel or RTOS
- Full context saves because an ISR may not return to the same task
- Library functions need to be reentrant
- ISRs need to be task aware



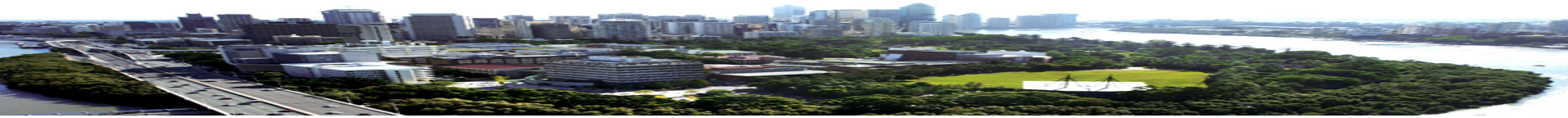
Reentrant?

- A task makes a function call. An interrupt occurs and there is a context switch. Next time the task executes and the function resumes, is everything the same?
- Local variables saved on stack frames should be used.



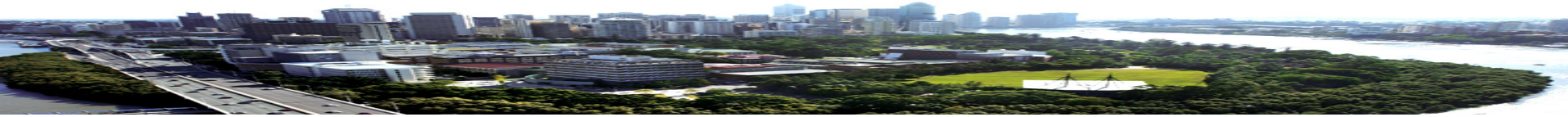
A reentrant function

```
void strcpy(char *dest, char *src)
{
   while (*dest++ = *src++);  // goes until NULL
   *dest = NULL; // insert the NULL char also
}
/* Pointers passed are saved on stack, These
   are incremented. Next entry will resume from
   where it was left. Next call will start over. */
```



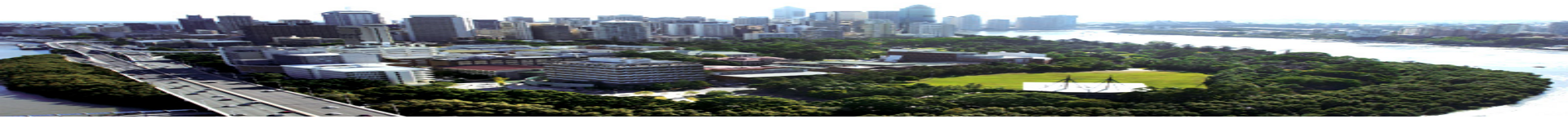
A non-reentrant function

```
int Temp;                          // this is global
void swap (int *x, int *y)
{
Temp = *x;
*x = *y;                           // if interrupted here?
*y = Temp;
}
// Temp may be changed  when entered again
```



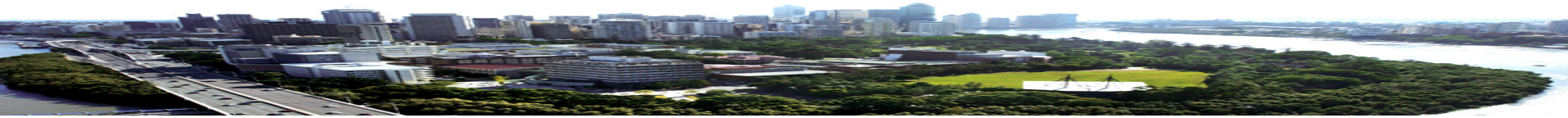
Task aware?

- A task tries to read from an input buffer and is suspended because the buffer is empty. It is waiting on this event and there may be a queue of such waiting tasks. An input ISR places input into the buffer. It also needs to change the state of the waiting task(s) to ‘ready to run’.



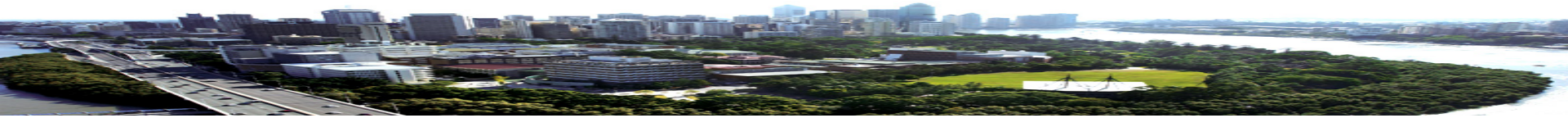
RTOS

- Real time Operating System (RTOS)
  - routines to create tasks
  - Routines for inter-task communication through message buffers
  - Task prioritization and scheduling
  - basic input output ISRs
  - synchronization and mutual exclusion primitives such as semaphores
  - mechanisms for user written ISRs
  - Can be enhanced to add a memory manager



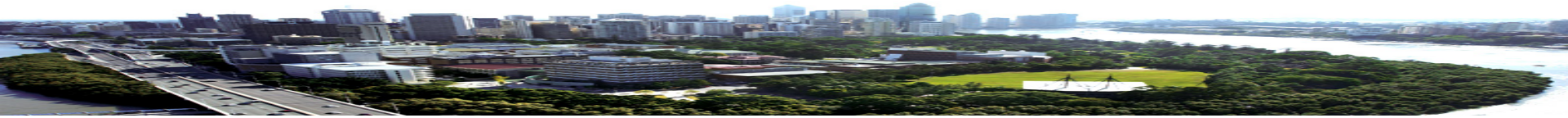
Examples

- Micro C OS/II (ucos-II) used with Dynamic C in the laboratory is a pre-emptive microkernel RTOS
- Dynamic C itself functions as an RTOS with co-operative (yield) and pre-emptive (slice) multitasking capabilities
- RT-Linux, TRON, ThreadX, QNX, Nucleus, Fusion, LynxOS, ChibiOS/RT, FreeRTOS etc. are among many others available



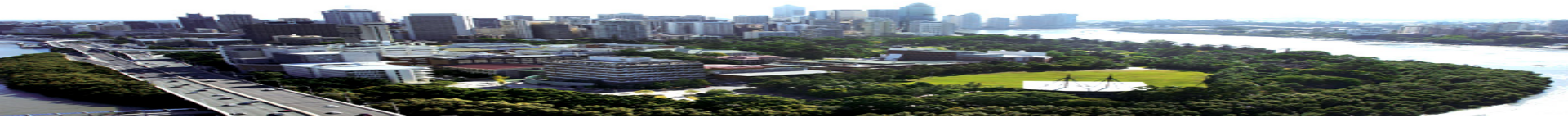
When to use co-operative multitasking

- When the programmer can have complete control over all tasks through knowledge of the system operation
- When loop times are likely to be low
- When a lot of work is done with some data and many tasks are reading and updating it frequently (task inter-dependence is high)

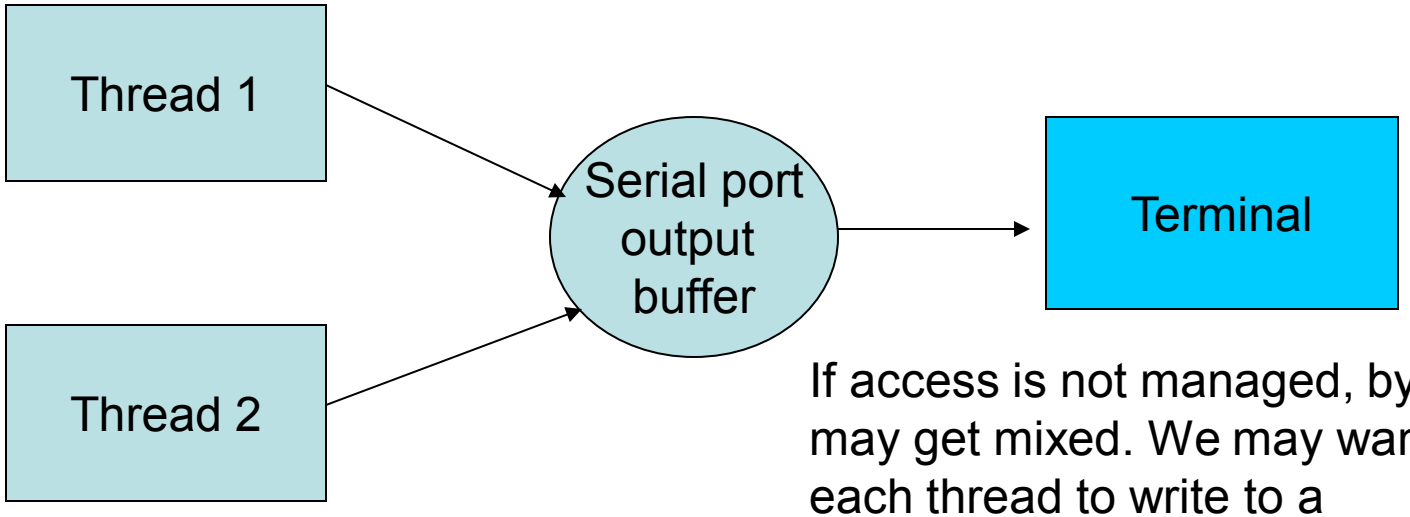


When to use pre-emptive multitasking

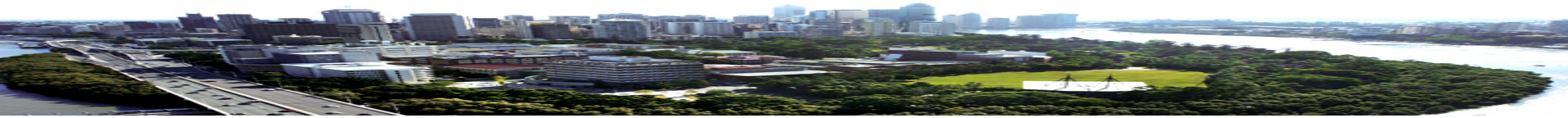
- When tasks are fairly independent and do not share a lot of resources
- When response time is critical and tasks must be prioritized
- If the loop time is high for tasks.



Access to a shared resource

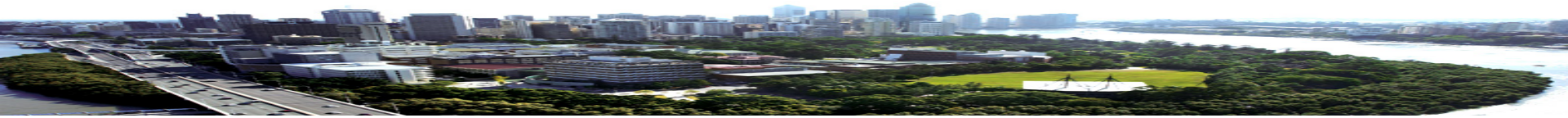


If access is not managed, bytes may get mixed. We may want each thread to write to a separate window and preserve message boundaries. Shared resources will be covered fully in the next lecture.



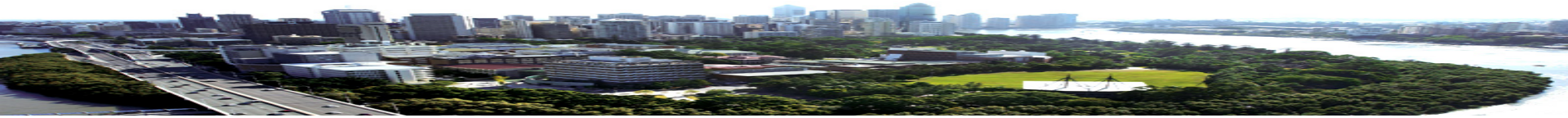
Inter-task communication

- Tasks may not share a common address space. They need not be compiled and linked together. They have their own stacks.
- To communicate (pass a data value for example) they need to go through the RTOS.
- RTOS provides mechanisms such as mailboxes, queues, events, signals.



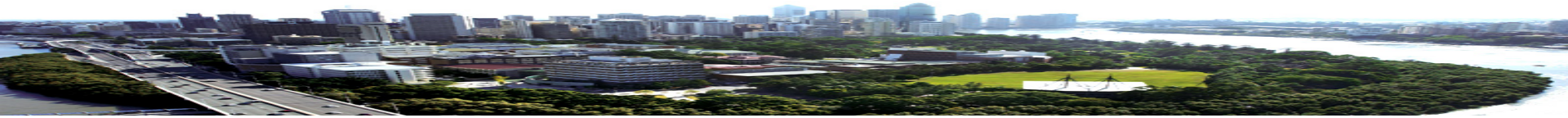
Dynamic C and pre-emptive multitasking

- No direct support in Dynamic C for task prioritization. Indirectly, time slices of unequal length may be used.
- No direct support for inter-task communication in Dynamic C. Global variables
- Dynamic C allows the RTOS ucos-II as a library which provides the above.



Micro C/OS-II

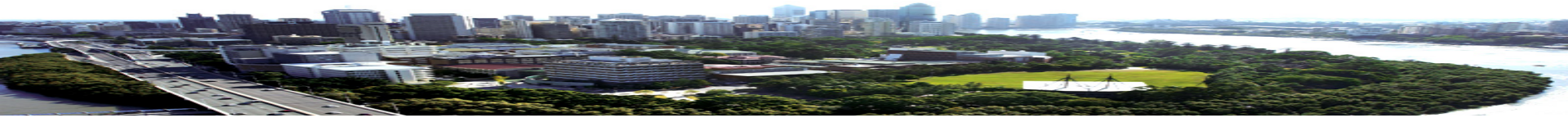
- Reference book – Micro C/OS-II: The Real-time Kernel by Jean J. Labrosse, CMP books ISBN 1-57820-103-9
- Chapter 1 – Getting Started with uC/OS-II explains usage through example programs. These examples are available in the Samples folder and a couple are used for demonstration. Good place to start.
- The entire book and volume of information is daunting. It is not required for this class. For the assignment you have a choice of cooperative or pre-emptive multi-tasking. For lab exercises 5 and 6 you will use this RTOS and the book serves as a reference.
- There are a few copies in the lab S1129. Please use them and leave them in the lab. Help on the usage of functions can also be obtained from within Dynamic C.



Example programs

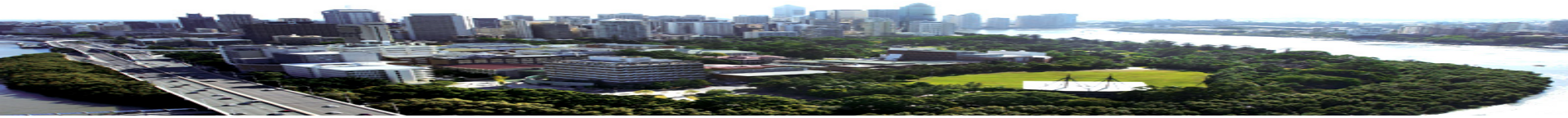
- UcosDemo1.c : Demonstrates basic multtasking capabilities of uC/OS-II
- UcosDemo2.c : Demonstrates stack checking features
- UcosDemo3.c : Demonstrates task execution time. There is a statistics task already implemented.
- UcosCoord.c : Demonstrates a number of features and real-time concepts. This program is used for demonstration in the lecture.
- Ucostaisr.c : An edited version is used in the demonstration for lecture 10 and shows use of semaphores.

Concepts such as semaphores and mailboxes will be covered in the next lecture. Real-time concepts such as deadlines and scheduling will be covered in lecture 10.



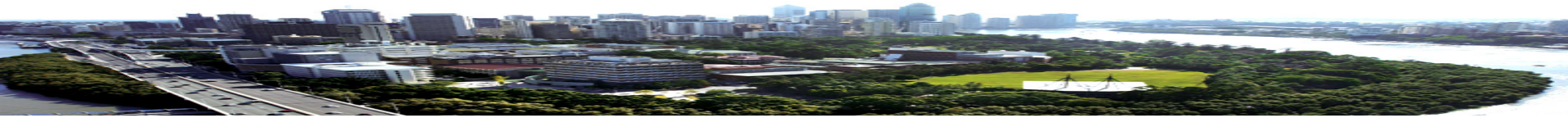
OSTaskCreate

```
OSTaskCreate                                                    <UCOS.LIB>

SYNTAX:  char OSTaskCreate(void (*task)(),void *pdata,INT16U stk_size,
                           char priority);

DESCRIPTION:  This function is used to have uC/OS-II manage the execution
of a task. Tasks can either be created prior to the start of multitasking
or by a running task.  A task cannot be created by an ISR.

PARAMETER1: pointer to task function
PARAMETER2: pointer to an optional data area which can be used to
pass parameters to the task when the task first executes.  Where the task
is concerned it thinks it was invoked and passed the argument 'pdata' as
follows:

    void Task (void *pdata){for (;;) { Task code ;}  }

PARAMETER3:  size of task's stack in bytes.

PARAMETER4: the task's priority.  A unique priority ( 0-62 ) MUST be
assigned to each task and the lower the number, the higher the priority.
(63 is for idle task)

RETURN VALUE: OS_NO_ERR        if the function was successful.
              OS_PRIO_EXIT     if the task priority already exist
                               (each task MUST have a unique priority).
              OS_PRIO_INVALID  if the priority specified is higher than
                               the maximum allowed (i.e. >= OS_LOWEST_PRIO)
```



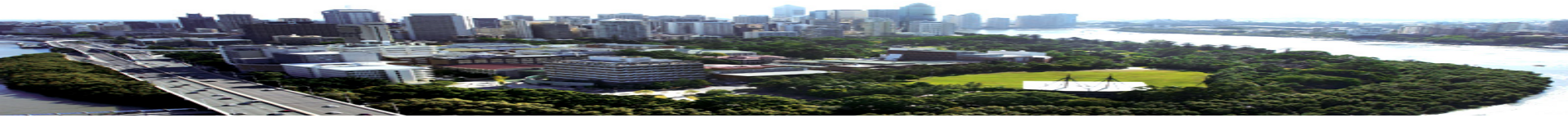
OSQCreate

```
OSQCreate                                                    <UCOS.LIB>

SYNTAX:         OS_EVENT *OSQCreate (void **start, INT16U qsize);

DESCRIPTION:    This function creates a message queue if free event control
                blocks are available.

PARAMETER1:     Pointer to the base address of the message queue storage
                area.  The storage area MUST be declared as an array of
                pointers to 'void' as follows:
                     void *MessageStorage[qsize]

PARAMETER2:     Number of elements in the storage area

RETURN VALUE:   != (void *)0  is a pointer to the event control clock
                              (OS_EVENT) associated with the created queue
                == (void *)0  if no event control blocks were available
```



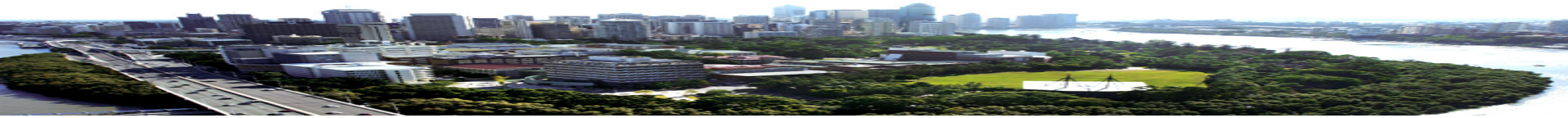
OSMemCreate

```
OSMemCreate                                                    <UCOS.LIB>

SYNTAX:        OS_MEM *OSMemCreate (void *addr, INT32U nblks,
                                    INT32U blksize, INT8U *err);

DESCRIPTION:   Create a fixed-sized memory partition that will be managed
               by uC/OS-II.

PARAMETER1:    The starting address of the memory partition.

PARAMETER2:    The number of memory blocks to create from the partition.

PARAMETER3:    The size (in bytes) of each block in the memory partition.

PARAMETER4:    Pointer to a variable containing an error message which will
               be set by this function to either:
               OS_NO_ERR             if the memory partition has been
                                     created correctly.
               OS_MEM_INVALID_ADDR   you are specifying an invalid address
                                     for the memory storage of the
                                     partition.
               OS_MEM_INVALID_PART   no free partitions available
               OS_MEM_INVALID_BLKS   user specified an invalid number of
                                     blocks (must be >= 2)
               OS_MEM_INVALID_SIZE   user specified an invalid block size
                                     (must be greater than the size of a
                                     pointer)

RETURN VALUE:  != (OS_MEM *)0   if the partition was created
               == (OS_MEM *)0   if the partition was not created because of
                                invalid arguments or, no free partition is
                                available.
```



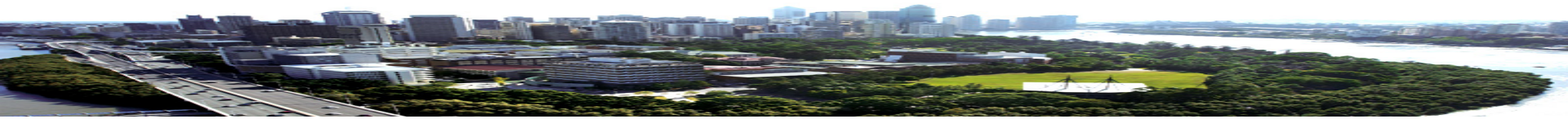
OSMBoxCreate

```
OSMboxCreate                                                    <UCOS.LIB>

SYNTAX:         OS_EVENT *OSMboxCreate (void *msg);

DESCRIPTION:    This function creates a message mailbox if free event
                control blocks are available.

PARAMETER1:     Pointer to a message to deposit in the mailbox.  If this
                value is set to the NULL pointer (i.e. (void *)0) then the
                mailbox will be considered empty.

RETURNS:        != (void *)0  is a pointer to the event control clock
                              (OS_EVENT) associated with the created mailbox
                == (void *)0  if no event control blocks were available
```



OSStart

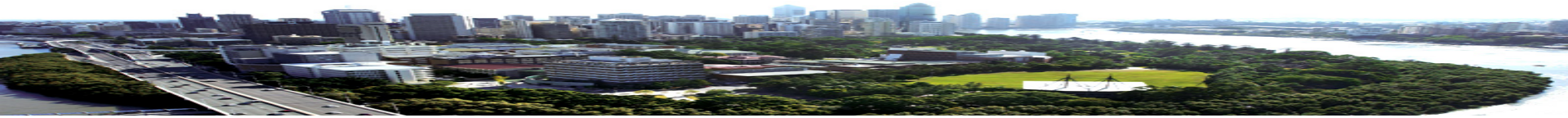
```
OSStart                                                          <UCOS.LIB>

SYNTAX:         void OSStart();

DESCRIPTION:    This function is used to start the multitasking process
                which lets uC/OS-II manage the tasks that you have created.
                Before you can call OSStart(), you MUST have called OSInit()
                and you MUST have created at least one task.  This function
                calls OSStartHighRdy which calls OSTaskSwHook and sets
                OSRunning to TRUE.
```



Ucos-II support for multitasking – code fragment

```
// Redefine uC/OS-II configuration constants as necessary

#define OS_MAX_EVENTS       2    // Maximum number of events (semaphores, queues, mailboxes)
#define OS_MAX_TASKS       11    // Maximum number of tasks system can create
#define OS_MAX_MEM_PART    10    // Maximum number of memory partions in system

#define OS_TASK_CREATE_EN   1    // Enable normal task creation
#define OS_MEM_EN           1    // Enable memory manager
#define OS_Q_EN             1    // Enable queues
#define OS_Q_POST_EN        1    // Enable pre 2.51 queue post method
#define OS_MBOX_EN          1    // Enable mailboxes
#define OS_MBOX_POST_EN     1    // Enable pre 2.51 mail box post method
#define OS_TICKS_PER_SEC   32    // Set the number of ticks in one second

#use "ucos2.lib"
```

For details of the microkernel – reference book : MicroC/OS-II by Jean Labrosse, CMP Books.



Ucos-II support for multitasking – code fragment

```
/ Function prototypes for tasks
void Master(void* pdata);
void Worker(void* pdata);

// Function prototypes for helper functions
void InitDisplay();
void DispStr(int x, int y, char *s);

void main()
{
        auto INT8U Error;
        auto INT16U Ticks;

        // Initialize uC/OS-II internal data structures
        OSInit();

        InitDisplay();

  // Create Master task
  Error = OSTaskCreate(Master, NULL, 512, 0);

  // Create queue for storing coordinates
        CoordinateQ = OSQCreate(&QStorage[0], 10);

        // Create and initialize uC/OS-II controlled memory partition.
        MsgMem = OSMemCreate(&MsgBuf[0][0], 10, 15, &Error);

  // Create mailbox tasks use to let Master know they are done moving
  DoneMbox = OSMboxCreate((void *)0);

  // Begin muli-tasking by entering the first ready task
        OSStart();
}
```

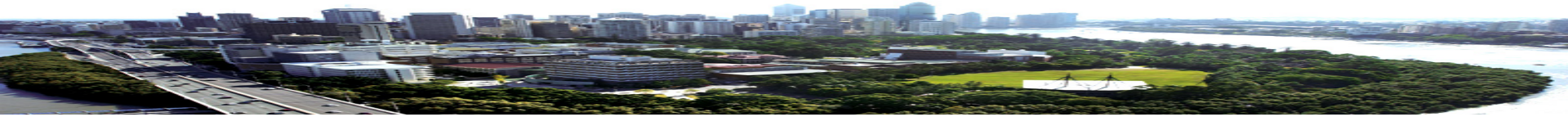
OS initialization

Task creation

Queueing of data

Memory management

Inter task communication



Demonstration

- Basic use of uCos/OS-II