CMR INSTITUTE OF TECHNOLOGY

Scheme Of Evaluation
Internal Assessment Test 2 – NOV.2016

Sub:	Embedded Computing Systems			**Code:**	10CS72
Date: 02/11/2016	Duration: 90mins	Max Marks: 50	**Sem:** VII	**Branch:**	CSE

Note: Answer Any Five Question

Question #		Description	Marks Distribution		Max Marks
1	a)	(i) Full workout of problem with correct NZCV ARM flag set (ii) Full workout of problem with correct NZCV ARM flag set	2 2	4	10 M
	b)	i. 1-2 differences plus at least 1 example ii. 1-2 differences plus at least 1 example iii. 1-2 differences plus at least 1 example	2 2 2	6	
2	a)	List at least 4 techniques Explain any one technique with example	2 2	4	10 M
	b)	i.Sample C code fragment for the above ARM assembly code ii.Lifetime graph iii.Modified C code statement iv.Lifetime graph for the modified C code v.ARM assembly code for the modified C code vi.DFG	1 1 1 1 1 1	6	
3	a)	i.Full Calculation of average memory access latency with correct answer ii. Full calculation of hit rate with correct answer	2 2	4	10 M
	b)	i. 1-2 differences plus at least 1 example ii. 1-2 differences plus at least 1 example iii. 1-2 differences plus at least 1 example	2 2 2	6	
4	a)	Explanation of user mode, and relevance to OS Explanation of Supervisor mode, and relevance to OS	2 2	4	10 M
	b)	i. Calculation of turn around time and waiting time using FIFO ii. Calculation of turn around time and waiting time using preemptive SJF iii. Calculation of turn around time and waiting time using RR Justification	1.5 1.5 1.5 1.5	6	

5	a)	Definition of RTOS List at least 6 services Explanation of at least one service in detail	2 1 1	4	10 M
	b)	i. ARM Assembly ii. CDFG iii. Correct calculation of cyclomatic complexity	2 2 2	6	
6	a)	Single assignment form DFG	2 2	4	10 M
	b)	Definition of IPC Explain at least 2 IPC techniques	2 4	6	
7	a)	i. Optimized code using code motion ii. optimized code using loop unrolling	2 2	4	10 M
	b)	Definition of Task Synchronization Explain at least 2 Task Synchronization techniques	2 4	6	
8	a)	Code logic and usage of POSIX primitives Syntax & Documentation	2 2	4	10 M
	b)	Functional Requirements Non-functional Requirements	3 3	6	

CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | | |

Internal Assesment Test - II

Sub:	Embedded Computing Systems							Code:	10CS72
Date:	02/11/2016	Duration:	90 mins	Max Marks:	50	Sem:	VII	Branch:	CSE

Answer Any FIVE FULL Questions

		Marks	OBE CO	OBE RBT
1(a)	Apply the principle of ARM status word control and calculate the CPSR status for the operations: (i) -1-1 (ii) 2^31 -1 +1	[4]	CO3	L3
(b)	Differentiate the following with examples: i. Cooperating vs Competing processes ii. DFG vs CDFG programming models iii. Deadlock vs Livelock	[6]	CO4	L4

(i) -1-1

-1: 1111……1111 (32 bits)

-1: 1111……1111 (32 bits)

On adding =(**1**)1111……1111 (32 bits + 1 carry out)

= -2 with carry out

N	Z	C	V
1	0	1	0

(ii) 2^31 -1 +1

2^31 -1: 0111….1111 (32 bits)

1: 0000….0001 (32 bits)

On adding=(**0**)1000….0000 (32 bits, NO carry out)

= -2^31

N	Z	C	V
1	0	0	1

i. Cooperating vs Competing processes

Cooperating processes	**Competing processes**
In this model, one process requires the inputs from other processes to complete its execution	In this model, the processes do not share anything among themselves, but they compete for the system resources.

They can further be classified as cooperation though sharing, and cooperation though communication	Here the classification in based on the type of resource being shared like file, display device etc
E.g. Process B requiring data X from Process A to execute.	E.g. Process A needs printer P to print file f1, and Process B also needs Printer P to print file f2.

ii. DFG vs CDFG programming models

DFG	**CDFG**
Data Flow Graph is a model of program with no conditionals.	Control/Data Flow Graph model uses DFG as an element, adding constructs to describe control.
Precisely, only one entry point and one exit point.	May have multiple exit points depending on how a program is written.
Constitutes one basic block consisting of operators as nodes and variables as edges.	Usually contains more than one basic blocks. Constitutes two types of nodes, viz., decision nodes and data flow nodes.
For the given input data set, ALL statements are executed.	Based on the input data set, only selected paths may execute.

e.g. C fragment:
`x=a+b;`
`y=a-c;`
`z=x+d;`
	e.g. C fragment:
	`i=0;`
	`if (a<b)`
	`  i+=10;`
	`else`
	`  i-=10;`

iii. Deadlock vs Livelock

Deadlock	**Livelock**
Condition in which a process is waiting for resource held by another process, which in turn is waiting for a resource held by the first process.	Condition in which a process always does something but is unable to make any progress towards execution completion.
Situation where none of the processes are able to make any progress in their execution due to the cyclic	Situation where progress seem to happen all the time but actually no real execution. This is similar to the

	dependency of resources. This ends up in none of the resources being utilized.	situation 'always busy, doing nothing'.			
	e.g. Pa blocked by Pb for resource. Pb blocked by Pa for resource.	e.g. Both Pa and Pb needs x and y for completion. step 1: Pa holds x, Pb holds y step 2: Pa drops x, Pb drops y Repeat steps 1 and 2			
(c)	--		[--]	--	--
2(a)	List the different program optimization techniques. Explain any one technique with an example. 1. Expression simplification 2. Dead code elimination 3. Procedure inlining 4. Loop transformations 5. Register allocation 6. Scheduling 7. Instruction selection **Expression Simplification:** This is a useful area for machine-independent transformations. Laws of algebra are used to simplify expressions. For example, distributive law is applied to rewrite the following expression: `a*b+a*c;` Re-written as `a*(b+c);` The new expression has only teo operations rather than three for the original form. This is certainly cheaper because it is both faster and smaller.		[4]	CO2	L2

(b)	Analyze the following ARM assembly code:
	
	`LDR r0,a`
	`LDR r1,b`
	`ADD r2,r0,r1`
	`STR r2,w`
	`LDR r0,c`
	`LDR r1,d`
	`ADD r2,r0,r1`
	`STR r2,x`
	`LDR r1,c`
	`ADD r0,r1,r2`
	`STR r0,u`
	`LDR r0,a`
			[6]	CO4	L4

```
LDR r1,b
SUB r2,r1,r0
STR r2,v
```

Answer the following:

i. Write the sample C code fragment for the above ARM assembly code

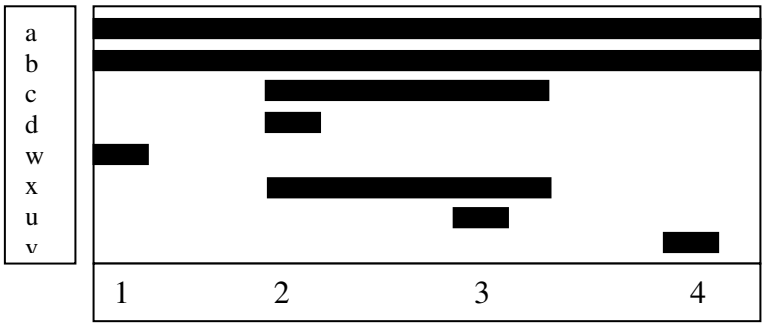
(1) w=a+b;

(2) x=c+d;

(3) u=c+x;

(4) v=b-a;

ii. Draw a lifetime graph that shows uses of register in register allocation from the above C statement.



iii. Modify the obtained C code statement using operator scheduling for register allocation

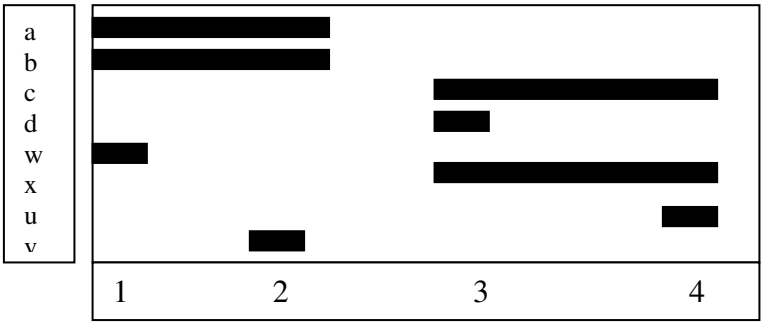
(1) w=a+b;

(2) v=b-a;

(3) x=c+d;

(4) u=c+x;

iv. Draw a lifetime graph for the modified C code



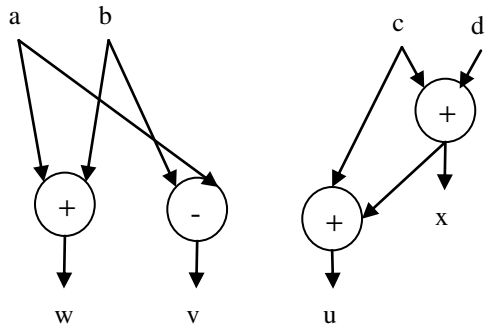
v. Write a ARM assembly code for the modified C code using register allocation.

```
LDR r0,a
LDR r1,b
ADD r2,r0,r1
STR r2,w
SUB r2,r1,r0
STR r2,v
LDR r0,c
LDR r1,d
ADD r2,r0,r1
STR r2,x
```

```
LDR r1,c
ADD r0,r1,r2
STR r0,u
```

vi.Draw DFG for (i) and (iii)

DFG for (i) and (iii) are the same, as shown below.



(c)	--	[--]	--	--
3(a)	Solve the following. i. What is the average memory access time of machine whose hit rate is 93% with cache access time of 5 nsec, main memory access time of 80 nano sec? Tav = h*Tcache + (1-h)*Tmain Tav = 0.93* 5 + (0.07)*80 ns Tav = 10.25 ns ii. Calculate cache hit rate if the cache access time is 5 nano sec, average memory access time is 6.5 nano sec, and main memory access time is 80 nano sec. Tav = h*Tcache + (1-h)*Tmain 6.5 = h*5 + (1-h)*80 h=0.98, or 98% hit rate	[4]	CO3	L3
(b)	Differentiate the following with examples: i. Counting semaphore vs Binary semaphore ii. Non-preemptive scheduling vs Preemptive Scheduling iii. Non-blocking vs Blocking communication	[6]	CO4	L4

i. Counting semaphore vs Binary semaphore

Counting semaphore	**Binary semaphore**
It is a sleep and wake up based mutual exclusion for shared memory access, which limits the access of resources by a fixed number of processes/threads.	Aka Mutex is also a counting semaphore, but restricting the access to only one process/thread at any given time.
Maintains count between 0 and N,	Uses 1-bit value tracking. That is

where N is the number of processes/threads that can access resource at a given time.	counts 0 to 1.
e.g Network card supporting N ports for N parallel communication.	e.g. Printer uses a 1-bit lock stating whether it is in use or not. A process can use a printer only if it is not locked.

ii. Non-preemptive scheduling vs Preemptive Scheduling

Non-preemptive scheduling	**Preemptive Scheduling**
In a multitasking model, this allows the currently executing task/process to run until it terminates or enters wait state, waiting for an IO or system resource.	In this multitasking model, every task in the Ready queue gets a chance to execute by evicting the currently running process. Here the scheduler temporarily pre-empts (stops) the currently running process.
E.g First Come First Served (FCFS), Last Come First served (LCFS), Shortest Job First (SJF), Priority based scheduling.	E.g. Preemptive SJF, Round Robin (RR).

iii. Non-blocking vs Blocking communication

Non-blocking	**Blocking communication**
Allows the process to continue execution after sending the communication.	After sending communication, the process goes to waiting state until it receives a response.
e.g. Synchronous RPC waiting for acknowledgement for every message sent.	e.g. Asynchronous RPC, where the calling process continues its execution while remote process executes the procedure.

(c) -- [--] -- --

4(a) Explain the user and supervisory mode structure in OS. [4] CO2 L2

The applications/services are classified into two categories: 1. User applications and 2. Kernel applications. The program code corresponding to the kernel applications/services are kept in contiguous area (OS dependent) of primary memory and is protected from the un-authorized access by the user programs/applications. The memory space at which the kernel code is located is known as kernel space. One way to access kernel space is via **supervisor mode** supported by the underlying architecture. For example, memory management systems allow the addresses of memory locations to be changed dynamically.

Control of the memory management unit (MMU) is typically reserved for supervisor mode to avoid the obvious problems that could occur when program bugs cause inadvertent changes in the memory management registers.

All user applications are loaded to a specific area of primary memory and this memory area is referred as User Space. All the user space applications run in **user mode**.

(b) Three processes with ID's P1, P2, P3 with estimated execution completion time 5, 10, 7ms respectively enters the ready queue together in the order P1, P2, P3. Process P4 with estimated execution completion time 2ms enters the ready queue after 5 ms. Which of the following scheduling policies is best for this scenario? Justify your choice. [6] CO3 L3

i. FIFO

P1	P2	P3	P4
0 5	6 15	16 22	23-24

Process	**Waiting Time (WT) (ms)**	**Turn Around Time (TAT) (ms)**
P1	0	5
P2	5	15
P3	15	22
P4	22-5 = 17	24-5=19
Average	**9.25**	**15.25**

ii. preemptive SJF

P1	P4	P3	P2
0 5	6 7	8 14	15 24

Process	**Waiting Time (WT) (ms)**	**Turn Around Time (TAT) (ms)**
P1	0	5
P2	14	24
P3	7	14
P4	5-5=0	7-5=2

Average	**5.25**	**11.25**

iii. RR (Time slice 2ms)

P1	P2	P3	P4	P1	P2	P3	P1	P2	P3	P2	P3	P2
0 2	3 4	5 6	7 8	9 10	11 12	13 14	15	16 17	18 19	20 21	22	23 24

Process	Waiting Time (WT) (ms)	Turn Around Time (TAT) (ms)
P1	10	15
P2	14	24
P3	15	22
P4	6-5=1	8-5=3
Average	**10.0**	**16.0**

From the above three scheduling policies, we see that preemptive SJF yields the least average waiting time (AWT) and Average Turn Around Time (ATAT). Thus, preemptive SJF is the best scheduling policy for the given scenario.

(c) -- [--] -- --

5(a) What is RTOS? List the different services of RTOS, and explain any one in detail. [4] CO2 L2

RTOS stands for Real-Time Operating System, which is a type of operating system that implements policies and rules concerning time-critical allocation of system resources. RTOS decides which applications should run in which order, and how much time needs to be allocated for each application. E.g. Windows CE, QNX, VxWorks MicroC/OS-II.

Services of RTOS:

1. Real-time Kernel
    a. Task/Process management
    b. Task/Process scheduling
    c. Task/Process synchronization
    d. Error/Exception handling
    e. Primary and Secondary Memory Management
    f. File System Management
    g. I/O system/ Device Management
    h. Interrupt Handling
    i. Time Management
    j. Protection systems

2. Hard Real-Time
3. Soft-Real-Time

Hard/Soft -Real time:
RTOS must adhere to the timing constraints of the processes/ applications. Based on the type of deadline, RTOS can be classified as either hard real time or soft real time system.
RTOS that strictly adhere to the deadline associated to tasks without any slippage are referred to as hard real-time systems. Missing any deadline may produce catastrophic results, including permanent data loss, irrecoverable damages and/or safety concerns. Here, the principle is '*A late answer is wrong answer*'. E.g. Anti-lock Braking System (ABS), Airbag control in vehicles.
RTOS that does not strictly guarantee meeting deadlines, but offers best effort to meet the deadline are referred to as soft real-time systems. Missing deadlines for tasks are acceptable if the frequency of missing deadline is within the compliance limit of the Quality of Service (QoS). E.g. Automatic Teller Machine (ATM), Audio-Video playback systems.

(b) Analyze the following C fragment: [6] CO3 L3

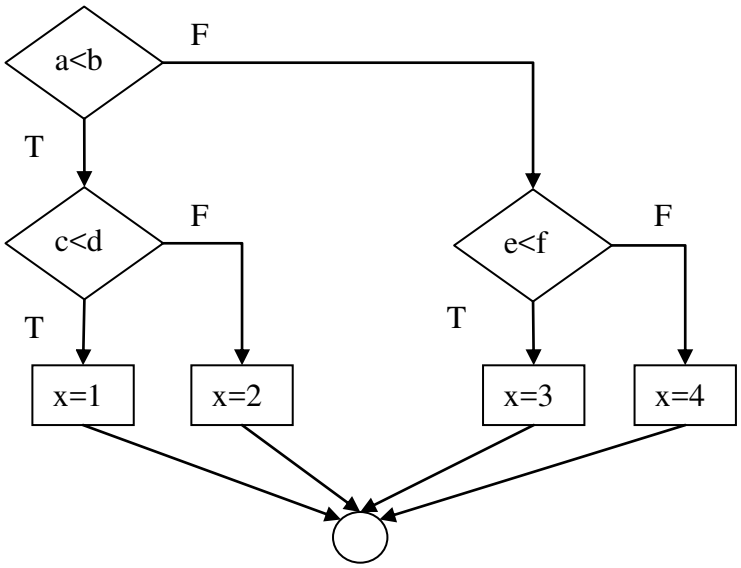
```
if (a < b) {
   if (c < d)
     x = 1;
   else
     x = 2;
}
else{
   if (e < f)
     x = 3;
   else
     x = 4;
}
```

i. Generate ARM assembly code.

```
ADR R0, a ;
LDR R1, [R0]; R1 <- a
ADR R0, b
LDR R2, [R0]; R2 <- b
ADR R0, c ;
LDR R3, [R0]; R3 <- c
ADR R0, d
LDR R4, [R0]; R4 <- d
CMP R1, R2
BGE outer_else
CMP R3, R4
BGE inner_else1
MOV R5, #1
JUMP after
inner_else1:MOV R5, #2
JUMP after
outer_else: ADR R0, e
LDR R3, [R0] ;  R3 <- e
```

```
ADR R0, f
LDR R4, [R0] ; R4 <- f
CMP R3, R4
BGE inner_else2
MOV R5, #3
JUMP after
inner_else2: MOV R5, #4
after: ADR R0, x
STR R5, [R0] ; R5 -> x
```

ii. Draw the CDFG.



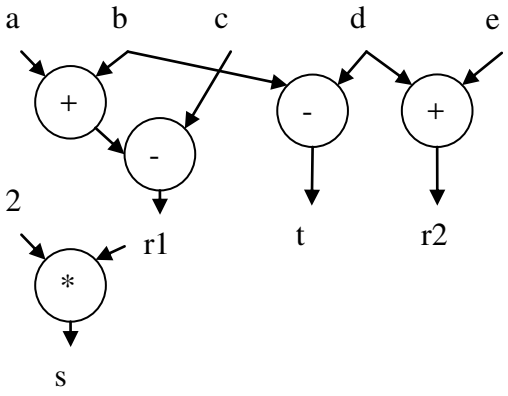
iii. Find the cyclomatic complexity of the CDFGs.

Cyclomatic Complexity M = e – n + 2p
From the above CDFG,
Number of edges, e = 10
Number of nodes, n= 8
Number of exit points, p = 1
Therefore, Cyclomatic Complexity M = 10-8+2=4

(c)	--	[--]	--	--

6(a)	For the following basic block given below, rewrite it in single-assignment form, and then generate the DFG for that form.
	`r=a+b-c;`
	`s=2*r;`
	`t=b-d;`
	`r=d+e;`
	
	Single-Assignment Form:
	
	`r1=a+b-c;`
	`s=2*r1;`
		[4]	CO3	L3

```
t=b-d;
r2=d+e;
```



(b) What is inter-process communication (IPC)? Explain the different IPC techniques. [6] CO2 L2

Inter Process Communication (IPC) is the mechanism provided by the OS as part of the process abstraction through which the processes/tasks communicate with each other.

Some of the important IPC mechanisms adopted by various kernels are explained below:

1. Shared Memory
   Processes share some area of the memory to communicate by the process is written to the shared memory area. Other processes which require this information can read the same from the shared memory area.

   Some of the different mechanisms adopted by different kernels are as below:
   a. Pipes: Pipe is a section of the shared memory used by processes for communicating. Pipes follow the client-server architecture. A process which creates a pipe is known as a pipe server and a process which connects to a pipe is known as pipe client. It can be unidirectional, allowing information flow in one direction or it can be bidirectional, allowing bi-directional information flow. Generally, there are two types of pipes supported by the OS. They are:
      Anonymous pipes: They are unnamed, unidirectional pipes used for data transfer between two processes.
      Names Pipes: They are named, unidirectional or bidirectional for data exchange between two processes.
   b. Memory Mapped Objects: This is a shared memory technique adopted by some real-time OS for allocating shared block of memory which can be accesses by multiple process simultaneously. In this approach, a

mapping object is created and physical storage for it is reserved and committed. A process can map the entire committed physical area ir a block of it to its virtual address space. All read-write operations to this virtual address space by a process is directed to its committed physical area. Any process which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for sharing the data.

2. Message Passing

   Message passing is an (a)synchronous information exchange mechanism used for Inter Process/Thread communication. The major difference between shared memory and message passing is that through shared memory lots of data can be shared whereas only limited amount of data is passed through message passing. Also, message passing is relatively fast and free from synchronization overheads compared to shared memory. Based on the message passing operation between the processes, message passing is classified into:

   a. Message Queue: Usually the process which wants to talk to another process posts the message to a First-In-First-Out (FIFO) queue called 'message queue', which stores the message temporarily in a system defined memory object, to pass it to the desired process. Messages are sent and received through *send* and *receive* methods. The messages are exchanged through the message queue. It should be noted that the exact implementation is OS dependent. The messaging mechanism is classified into synchronous and asynchronous based on the behavior of the message posting thread. In asynchronous messaging, the message posting thread just posts the message to the queue and it will not wait for an acceptance (return) from the thread to which the message is posted. Whereas in synchronous messaging, the thread which the message is posts the message enters waiting state and waits for the message result from the thread to which the message is posted. The thread which invoked the send message becomes blocked and the scheduler will not pick it up for scheduling.

   b. Mailbox: Mailbox is an alternative to 'message queues' used in certain RTOS for IPC, usually used for one way messaging. The thread which creates the mailbox is known as 'mailbox server' and the threads which subscribe to the mailbox are known as 'mailbox clients'. The mailbox server posts messages to the mailbox and notifies it to the clients which are subscribed to the mailbox. The clients read the message from the mailbox on receiving the notification. The process of creation, subscription, message reading and writing are achieved through OS kernel provided API calls.

c. Signaling: Signaling is a primitive way of communication between processes/threads. *Signals* are used for asynchronous notifications where one process/trhead fires a signal, indicating the occurrence of a scenario which the other process(es)/thread(s) is waiting. Signals are not queued and they do not carry any data.

3. Remote Procedure calls and Sockets

Remote Procedure Call (RPC) is the IPC mechanism used by a process to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network. In object oriented language terminology RCP is also known as *Remote Method Invocation (RMI)*. RPC is mainly used for distributed applications like client-server applications. The CPU/process containing the procedure which needs to be invoked remotely is known as server. The CPU/process which initiates an RPC request is known as client.

Sockets are used for RPC communication. Socket is a logical endpoint in a two-way communication link between two applications running on a network. Sockets are of different types, namely, Internet Sockets (INET), UNIX sockets, etc. The INET sockets works on internet communication protocols, such as TCP/IP and UDP. They are classified into stream sockets and datagram sockets.

Stream sockets are connection oriented, and they use TCP to establish a reliable connection.

Datagram sockets rely on UDP for communication. The UDP connection is unreliable when compared to TCP.

(c)	--	[--]	--	--
7(a)	Consider the following loop.	[4]	CO3	L3

```
int N=8, M=4;
for (i = 0; i < N*M; i++)
  x[i] = a[i] * c[i];
```

i. Optimize the code applying code motion technique.

```
int N=8, M=4;
temp = N*M;
for (i = 0; i < temp; i++)
  x[i] = a[i] * c[i];
```

ii. Optimize the code applying loop unrolling 2 times.

```
int N=8, M=4;
temp = N*M;
```

```
for (i = 0; i < temp; i+=2)
            x[i] = a[i] * c[i]
            x[i+1] = a[i+1] * c[i+1]
```

(b) What is Task synchronization? Explain the different Task synchronization techniques. [6] CO2 L2

The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as Task synchronization. Task synchronization is essential for 1. Avoiding conflicts in resource access (racing, deadlock, livelock, starvation) in a multitasking environment, and 2. Ensuring proper sequence of operation across processes. E.g. producer-consumer problem.

Different task synchronization techniques to address them are as follows:

1. Mutual Exclusion through busy waiting/spin lock:
   Busy waiting is the simplest method for enforcing mutual exclusion. The busy waiting technique uses a lock variable for implementing mutual exclusion. Each process/thread checks this lock variable before entering the critical section. The lock is set to '1' by a process/thread if the process/thread is already in its critical section; otherwise the lock is set to '0'. The major challenge in implementing the lock variable based synchronization is the non-availability of a single atomic instruction which combines the reading, comparing and setting of the lock variable. To address this issue is tackled by combining the actions of reading the lock variable, testing its state and setting the lock into a single step, with a combined hardware and software support. Most processors support a single instruction 'Test and Set Lock' (TSL) for testing and software support. This instruction call copies the value of the lock variable and sets it to a nonzero value.
   The lock based mutual exclusion implementation always checks the state of a lock and waits till the lock is available. This keeps the processes/threads always busy and forces the processes/threads to wait or spin in one state till the availability of the lock for proceeding further. Hence, this synchronization is got the name 'Busy Waiting' or 'Spin Lock'. For the same reason, this mechanism leads to underutilization, wastage of processor time and power consumption.

2. Mutual Exclusion through Sleep and Wake up:
   An alternative to 'busy waiting' is the 'Sleep & Wakeup' mechanism. When a process is not allowed to access the critical section that has been locked by another process, the process undergoes 'Sleep' and enters 'Blocked' state. The process which is blocked on waiting for access to the critical section is awakened by the process which currently owns the critical section. Sleep &

Wake can be implemented in different ways. Few of them are listed below.

Semaphores: It is a sleep and wake up based mutual exclusion for shared memory access, which limits the access of resources by a fixed number of processes/threads. This is further classified into two: *Binary semaphore* and *Counting Semaphore*. The binary semaphore, also called *mutex*, provides exclusive access to shared resource by allocating the resource to a single process at a time, and not allowing other process to access it when it is being owned by a process. Counting Semaphore, on the other hand, maintains a count between zero and a value. It limits the usage of a resource to the maximum value of the count supported by it.

Critical Section Objects: In Windows CE, the critical section object is same as the mutex object, except that Critical section object can only be used by the threads of a single process (Intra process). The piece of code which needs to be made critical section is places in the 'critical section' area by the process. The memory area which is to be used as the 'critical section' is allocated by the process. Once the critical section is initialized, all threads in the process can use it using an API call for getting exclusive ownership of the critical section.

Events: Event object is a synchronization technique which uses the notification mechanism for synchronization. In the concurrent execution we may come across situations which demand processes to wait for a particular sequence for its operations. For example, in producer-consumer threads, the consumer should wait to consume the data for producer to produce the data, and likewise, producer should wait for consumer to consume data. Event objects are helpful to implement notification mechanisms in such scenarios. A thread/process can wait for an event and another thread/process can set this event for processing by the waiting thread/process.

(c)	--	[--]	--	--
8(a)	Applying multithreading concept, write a program to print "Hello I'm main thread" from main thread, and "Hello I'm child thread" from child thread using POSIX primitives.	[4]	CO3	L3

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

//Child thread function
void *new_thread(void *thread_args)
{
  //Print from Child thread
```

```
  printf("Hello I'm child thread\n");
  return NULL;
}
//Main thread
int main(void)
{
  pthread_t tcb;
  //create child thread
  if(pthread_create(&tcb, NULL, new_thread, NULL))
  {
    //Child thread creation failed
    printf("Error creating thread!\n");
    return -1;
  }

  //Print from Main thread
  printf("Hello I'm main thread\n");

  //Join child thread here
  if(pthread_join(tcb,NULL))
  {
   //Joining failed
   printf("Error in joining thread!\n");
   return -1;
  }
return 1;
}
```

(b) Explain all the factors that need to be evaluated in selection of an RTOS. [6] CO2 L2

The decision of choosing RTOS for an embedded design is very critical. A lot of factors need to be carefully analyzed before making the decision on the selection of an RTOS. These can be either functional or non-functional..

1. Functional Requirements
   a. Processor Support: It is not necessary that all RTOS support all kinds of processor architecture. It is essential to ensure the processor support by the RTOS.
   b. Memory Requirements: The OS requires ROM memory for holding the OS files and it is normally stored in a non-volatile memory like FLASH. OS also requires working memory RAM for loading the OS services. Since embedded systems are memory constrained, it is essential to evaluate the minimal ROM and RAM requirements for the OS under consideration.
   c. Real-time capabilities: It is not mandatory that the OS for all embedded systems need to be real-time and all embedded systems are 'real-time' in behavior. The task/process scheduling policies plays an important role in the 'real-time' behavior of an OS. Analyze the real-time capabilities if the OS under consideration and the standards met by the OS for real

time capabilities.

d. Kernel and Interrupt Latency: The kernel of the OS may disable interrupts while executing certain services and it may lead to interrupt latency. For and embedded system whose response requirements are high, this latency should be minimal.
e. Inter Process Communication and Task Synchronization: The implementation of IPC and Synchronization is OS kernel dependent. Certain kernels may provide a bunch of options whereas others provide very limited options. Certain kernels implement policies for avoiding priority inversion issues in resource sharing.
f. Modularization support: Most of the OS provide a bunch of features. At times it may not be necessary for an embedded product for its functioning. It is very useful if the OS supports modularization in the developer can choose the essential modules and re-compile the OS image for functioning. E.g. Windows CE.
g. Support for Networking and Communication: The OS kernel may provide stack implementation and driver support for a bunch of communication interfaces and networking. Ensure that the OS under consideration provides support for all the interfaces required by the embedded product.
h. Development and Debugging Support: Certain OS include runtime libraries required for running applications written in languages like Java and C#. A Java Virtual Machine (JVM) customized for the OS is essential for running java applications.

2. Non-Functional Requirements
   a. Custom Developed or Off the Shelf: Depending on the OS requirement, it is possible to go for the complete development of an OS suiting the embedded system needs or use an off the shelf, readily available OS, which is either a commercial product or an Open Source product, which is in close match with the system requirements. Sometimes it may be possible to build the required features by customizing the Open Source OS. The decision is purely dependent on the development cost, licensing fees for the OS, development time and availability of skilled resources.
   b. Cost: The total cost for developing or buying the OS and maintaining it in terms of commercial product and custom build needs to be evaluated before taking a decision on the selection of OS.
   c. Development and Debugging Tools Availability: The availability of development and debugging tools is a critical decision making factor in the selection of an OS for embedded design. Certain Operating Systems may be superior in performance, but the availability of tools for supporting the development may be limited.
   d. Ease of Use: How easy is it to use a commercial RTOS is another

	important feature that needs to be considered in the RTOS selection. e. After Sales: For a commercial embedded RTOS, after sales in the form of e-mail, on-call services, etc for bug fixes, critical patch updates and support for production issues, etc should be analyzed thoroughly.			
(c)	--	[]	--	--

Course Outcomes		PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1:	Identify and describe the hardware & software components and functional & non-functional features of embedded systems.	1	-	-	-	-	-	-	-	-	-	-	-
CO2:	Explain the functionalities and various challenges faced in embedded computing.	1	1	-	-	-	-	-	-	-	-	-	-
CO3:	Apply the principles of system design process and implement each design phase.	2	1	-	-	1	-	-	-	-	-	-	-
CO4:	Analyze existing embedded system applications, and their relationship between different hardware and software components.	2	3	-	2	2	2	2	-	1	1	-	-
CO5:	Test designs at different levels using verification and validation techniques.	2	2	-	2	3	-	-	-	1	1	-	-
CO6:	Design solutions to overcome limitations in existing embedded system application.	3	3	3	3	3	2	2	2	3	3	2	2

Cognitive level	KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.

PO1 - *Engineering knowledge*; PO2 - *Problem analysis*; PO3 - *Design/development of solutions*; PO4 - *Conduct investigations of complex problems*; PO5 - *Modern tool usage*; PO6 - *The Engineer and society*; PO7- *Environment and sustainability*; PO8 – *Ethics*; PO9 - *Individual and team work*; PO10 - *Communication*; PO11 - *Project management and finance*; PO12 - *Life-long learning*

GOOD LUCK!