UNIT IV

RTOS BASED EMBEDDED SYSTEM DESIGN

4.3 Memory Management

Functions Memory allocation

- When a process is created ,the memory manager allocates the memory addresses (blocks) to it by mapping the process address space.
- Threads of a process share the memory space of the process

- Memory manager of the OS ─ secure , robust and well protected.
- No memory leaks and stack overflows
- Memory leaks means attempts to write in the memory block not allocated to a process or data structure.
- Stack over flow means that the stack exceeding the allocated memory block(s)

Memory Management after Initial Allocation

Memory Managing Strategy for a system

- Fixed-blocks allocation
- Dynamic-blocks Allocation

- Dynamic Page - Allocation
- Dynamic Data memory Allocation
- Dynamic address - relocation
- Multi processor Memory Allocation
- Memory Protection to OS functions

Memory allocation in RTOS

- RTOS may disable the support to the dynamic block allocation, MMU support to dynamic page allocation and dynamic binding as this increases the latency of servicing the tasks and ISRs.
- RTOS may not support to memory protection of the OS functions as this increases the latency of servicing the tasks and ISRs.
- User functions are then can run in kernel space and run like kernel functions
- RTOS may provide for disabling of the support to memory protection among the tasks as this increases the memory requirement for each task

Memory Manager functions

(i) use of memory address space by a process,
(ii) specific mechanisms to share the memory space and
(iii) specific mechanisms to restrict sharing of a given memory space
(iv) optimization of the access periods of a memory by using an hierarchy of memory (caches primary and external secondary magnetic and optical memories).
Remember that the access periods are in the following increasing order : caches, primary and external secondary magnetic and then or optical.

Fragmentation Memory Allocation
Problems
Fragmented not continuous memory addresses in two blocks of a process

- Time is spending first locating next free memory address before allocating that to the process.
- A standard memory allocation scheme is to scan a linked list of indeterminate length to find a suitable free memory block.
- When one allotted block of memory is de allocated the time is spent in first locating next allocated memory block before de allocating that to the process.
- the time for allocation and de-allocation of the memory and blocks are variable (not deterministic) when the block sizes are variable and when the memory is fragmented.
- In RTOS, this leads to un predictable task performance

Memory management Example

RTOSCOS-II

- Memory partitioning
- A task must create a memory partition or several memory partitions by using function Mem Create ()
- Then the task is permitted to use the partition or partitions.
- A partition has several memory blocks.
- Task consists of several fixed size memory blocks.
- The fixed size memory blocks allocation and de-allocation time takes fixed time (deterministic).
- OS Mem Get()

—to provide a task a memory block or blocks from the partition

- OS Mem Put ( )

—to release a memory block or blocks to the partition

Interrupt Service routine

- ISR is a function called on an interrupt from an interrupting source.
- Further un like a function , the ISR can have hardware and software assigned priorities.
- Further unlike a function , the ISR can have mask, which inhibits execution on the event, when mask is set and enables execution when mask reset.

Task

- Task defined as an executing computational unit that processes on a CPU and state of which is under the control of kernel of an operating system.

Distinction Between Function, ISR and Task Uses

- Function ─ for running specific set of codes for performing a specific set of actions as per the arguments passed to it
- ISR ─ for running on an event specific set of codes for performing a specific set of actions for servicing the interrupt call.
- Task ─ for running codes on context switching to it by OS and the codes can be in endless loop for the event (s)

Calling Source

- Function ─ call from another function or process or thread or task.
- ISR─ interrupt-call for running an ISR can be from hardware or software at any Instance.
- Task ─ A call to run the task is from the system (RTOS). RTOS can let another higher priority task execute after blocking the present one. It is the RTOS (kernel) only that controls the task scheduling.

Context Saving

- Function─ run by change in program counter instantaneous value. There is a stack. On the top of which the program counter value (for the code left without running) and other values (called functions context) save.
- All function have a common stack in order to support the nesting
- ISR─ Each ISR is an event-driven function code. The code run by change in program counters instantaneous value. ISR has a stack for the program counter instantaneous value and other values that must save.
- All ISRs can have common stack in case the OS supports nesting
- Task ─ Each task has a distinct task stack at distinct memory block for the context (program counter instantaneous value and other CPU register values in task control block) that must save.
- Each task has a distinct process structure (TCB) for it at distinct memory block

Response and Synchronization

- Function ─ nesting of one another, a hardware mechanism for sequential nested mode synchronization between the functions directly without control of scheduler or OS
- ISR─ a hardware mechanism for responding to an interrupt for the interrupt source calls, according to the given OS kernel feature a synchronizing mechanism for the ISRs, and that can be nesting support by the OS.
- ISR─ a hardware mechanism for responding to an interrupt for the interrupt source calls, according to the given OS kernel feature a synchronizing mechanism for the ISRs and that can be nesting support by the OS

Structure

- Function─ can be the subunit of a process or thread or task or ISR or subunit of another function.
- ISR─ Can be considered as a function, which runs on an event at the interrupting source.
- A pending interrupt is scheduled to run using an interrupt handling mechanism in the OS, the mechanism can be priority based scheduling.
- The system, during running of an ISR, can let another higher priority ISR run.
- Task ─ is independent and can be considered as a function, which is called to run by the OS scheduler using a context switching and task scheduling mechanism of the OS.
- The system, during running of a task, can let another higher priority task run. The kernel manages the tasks scheduling

Global Variables Use

- Function─ can change the global variables. The interrupts must be disabled and after finishing use of global variable the interrupts are enabled.
- ISR─ When using a global variable in it, the interrupts must be disabled and after finishing use of global variable the interrupts are enabled (analogous to case of a function).
- Task ─ When using a global variable, either the interrupts are disabled and after finishing use of global variable the interrupts are enabled or use lock functions in critical sections, which can use global variables and memory buffers.

Posting and Sending Parameters

- Function─ can get the parameters and messages through the arguments passed to it or global variables the references to which are made by it. Function returns the results of the Operations.
- ISR─ using IPC functions can send (post) the signals, tokens or messages. ISR can‘t use the mute protection of the critical sections by wait for the signals, token or messages.

- Task ─ can send (post) the signals and messages.
- can wait for the signals and messages using the IPC functions, can use the mute or lock protection of the code section by wait for the token or lock at the section beginning and messages and post the token or unlock at the section end.

Semaphore as an event signaling variable or notifying variable

- Suppose that there are two trains.
- Assume that they use an identical track.
- When the first train A is to start on the track , a signal or token for A is set (true taken)and
- Same signal or token for other train, B is reset (false, not released).

OS Functions for Semaphore as an event signaling variable or notifying variable:

- OS Functions provide for the use of a semaphore for signaling or notifying of certain action or notifying the acceptance of the notice or signal.
- Let a binary Boolean variable, *s*, represents the semaphore. The taken and post operations on s ─ *(I)* signals or notifies operations for communicating the occurrence of an event and (ii) for communicating taking note of the event.
- Notifying variable *s* is like a token ─ (i) acceptance of the token is taking note of that event (ii) Release of a token is the occurrence of an event

Binary Semaphore

- Let the token (flag for event occurrence)*s* initial value = 0
- Assume that the *s* increments from 0to1 for signaling or notifying occurrence of an event from a section of codes in a task or thread.
- When the event is taken note by section in another task waiting for that event, the *s* Decrements from 1 to 0 and the waiting task codes start another action.
- When s = 1─ assumed that it has been released (or sent or posted) and no task code section has taken it yet.
- When s=0─assumed that it has been taken (or accepted) and other task code
- section has not taken it yet

Binary Semaphore use in ISR and Task

- An ISR can release token.
- A task can release the token as well accept the token or wait for taking the token