CS 5372
Specification and Design of Real-Time Systems

Lecture 7:
Real-Time I/O, Partitioning, and Memory

1

Outline

- Quiz (15 min)
- Lecture: I/O + Memory (35)
- Lab Exercise: vxWorks Memory (20)
- Project Work (40)
- Exam Makeup Questions (60)

2

Issues

- Input / Output (I/O) concept and operations
- Communicating with devices
- Memory types
- Partitioning and protection
- Memory management
- Shared memory
- Memory allocation and fragmentation
- Memory in VxWorks
- Memory problems

3

Input-Output (I/O) Subsystem

- Variety of I/O devices require non-conventional device drivers
- I/O operations paradigm:
  - Communicate with device
  - Initiate I/O request
  - Transfer data
  - Notify operation completion
- Software engineer must understand the physical properties (register definition, operations, access method)
- The I/O incorporates all issues real-time software developer must face (timing, concurrency, synchronization, communications, interrupts)

4

Basic I/O Concepts

- I/O device hardware may include serial, network, wireless, hard drives, scanners, sensors, actuators, etc.
- From programmer viewpoint, each device is represented as set of registers that need to have specific address to allow for an access
- **Port-Mapped** vs. **Memory-Mapped** I/O:
  - There may be separate I/O address space or devices will have dedicated system address space
- **Character** Mode vs. **Block** Mode:
  - Data transfer can occur one character at a time or as entire block of data

5

Basic I/O Refresher

- Devices are defined by a file descriptor *fd* which is a small integer returned by *open()* or *create()*
- The created device (or file) is opened:
  - *fd = open ("name", flags, mode);*
  - Flags and mode are defining read/write features and permissions
- We read data from *fd* to the *buffer*:
  - *nBytes = read (fd, &buffer, maxBytes);*
- We write data to *fd* from the *buffer*:
  - *write (fd, &buffer, nBytes);*
- The buffer must be allocated first:
  - *buffer = malloc(maxBytes);*

6

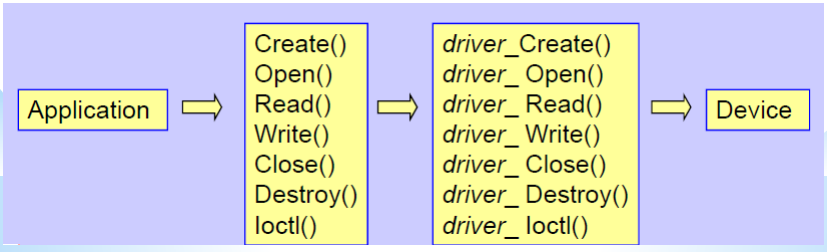
Basic I/O Operations

- Sequence of operations:
  - Communicating with device
  - Programming device to initiate I/O request
  - Actual data transfer
  - Notification of completion
- Seven basic I/O calls:
  - *create( )* - create a file/device
  - *remove( )* - remove a file/device
  - *open( )* - open a file/device (sometimes used to create)
  - *close( )* - close a file/device (sometimes used to delete)
  - *read( )* - read previously created or opened file/device
  - *write( )* - write previously created or opened file/device
  - *ioctl( )* - perform special control functions on file/device

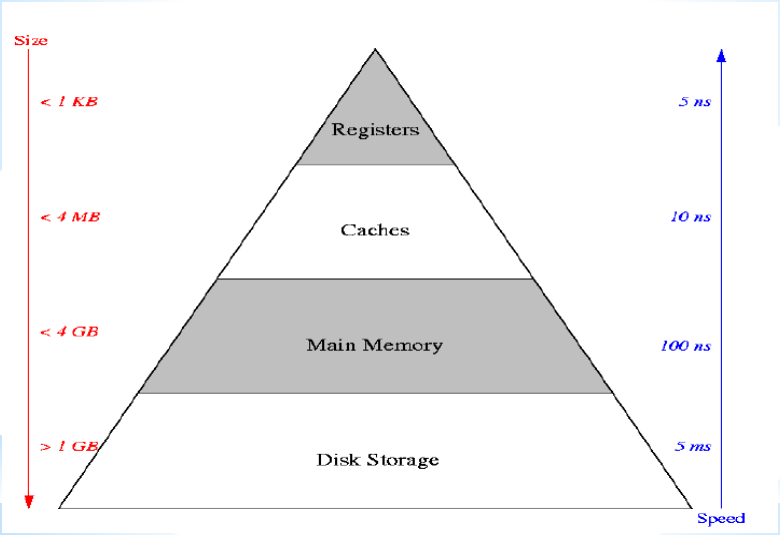
7

Mapping I/O Calls to Driver Calls

- The individual device driver provides implementation of basic I/O functions in a uniform API set
- The kernel driver table associates the devices with the device drivers (e.g. *driver* can be ***tty***, ***fei***, ***fd*** …)
- Mapping is initializing each generic I/O function pointer with the device specific address of driver function
- The record is kept in I/O driver table with entries created when driver is installed



Memory Hierarchy



9

What is Partitioning?

- Idea: to prevent undesirable interference between software components
- **Partitioning** is a functional separation of the applications
- Used primarily for fault containment: to prevent any partitioned application from causing a failure in another partitioned application
- **Partition** is a unit of partitioning, i.e. an equivalent of a stand-alone program in single application environment, consisting of data context related devices and data, context, related devices, configuration attributes
- However, we may need to establish multiple functional partitions within one program

10

Why do we need partitioning?

- Increasing interactions in the integrated embedded systems architectures
- ***Fault containment is the main objective of partitioning***
- The goal is to control the hazards created when a function shares resources with other functions
- Protection from loss of function is addressed by redundancy

11

Partitioning Options (1)

- **Physical partitioning** (in hardware):
  - involves the allocation of software components to hardware based on the hazard level of the components (only components of identical or similar hazard levels)
- **Logical partitioning** (in software):
  - used when physical partitioning is not possible
  - it requires the identification and appropriate protection of all resources that the software component may share with any other software component (examples?)

12

Partitioning Options (2)

- **Spatial partitioning** (in location):
  - to ensure that software in one partition cannot change the software or data of another partition, nor command the private devices or actuators of other partitions.
- **Temporal partitioning** (in time):
  - to ensure that the service received from shared resources by the software in one partition cannot be affected by the software in another partition (including rate, latency, duration, jitter)

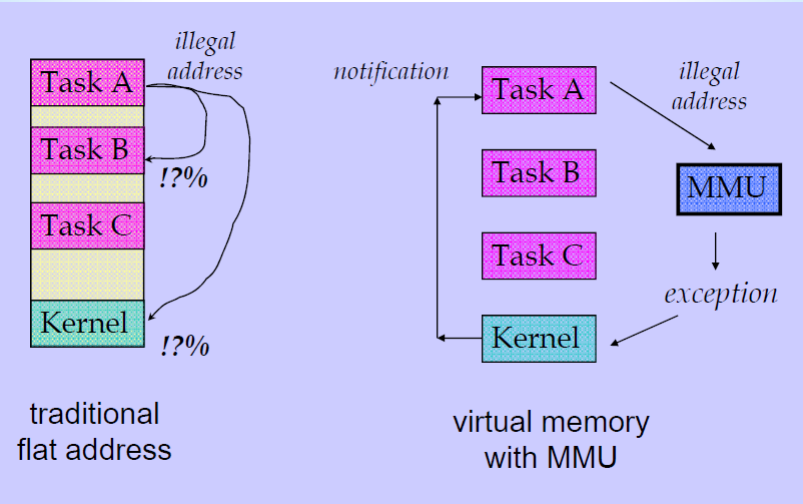
13

Spatial Partitioning Issues

- Communication – kernel executes copy from a buffer in one partition to another
  - An alternative might be a single buffer allocation that one partition can write and another read
  - Bi-directional buffer might be acceptable, but a separate buffer is required for each partition pair
  - Restrict unnecessary ***direct*** communication
- Naming conventions for partition address:
  - Absolute (e.g. partition A) – too rigid and limiting
  - Functional (e.g. Autopilot) – limits possible reconfiguration
  - Relative (e.g. port 13) – allows binding

14

Example: Spatial Protection



15

Temporal Partitioning Issues

- Concerns: monopolizing processor, halting processor, schedule overruns
- Runaway execution may be prevented by using ???
- but the other partitions may still be doomed (Example?)
- Notification of failure is required

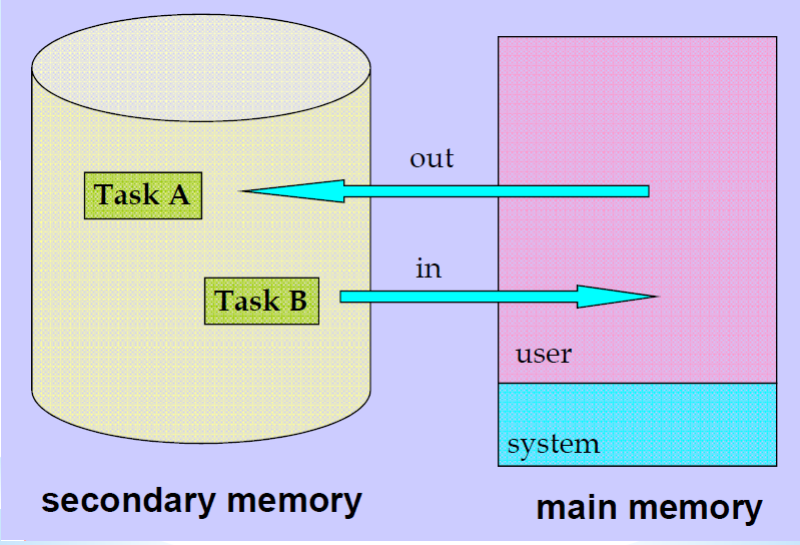
16

Memory Management: Observations

- Task must be in main memory to be executed
- Task may be swapped (rolled) in/out of main memory
- Only idle task can be swapped
- Swapping generally not a good idea for – real-time (Why?)

17

Example: Memory Swapping



18

Memory Management (MM) Approaches

- **Hardware MM** – concerned with the physical devices storing data including the access to ROM and processor cache
- **Operating System MM** – handling the allocation of memory to application programs and the translation between the virtual and physical memories
- **Application MM** – involves allocation and de-allocation of memory to the programming constructs (buffers, queues, data structures)

19

Shared Memory - Basics

- Shared memory and memory mapping are extremely low level communication mechanisms
- They require great deal of synchronization to be implemented within the application (mutex, semaphore, condition variable, monitor)
- The advantage is great speed – the processes "see" the memory immediately
- POSIX_SHARED_MEMORY_OBJECTS option is required (see unistd.h)
- Additionally, _POSIX_MAPPED_FILES and _POSIX_SYNCHRONIZED_IO are required to use memory mapping and synchronization

20

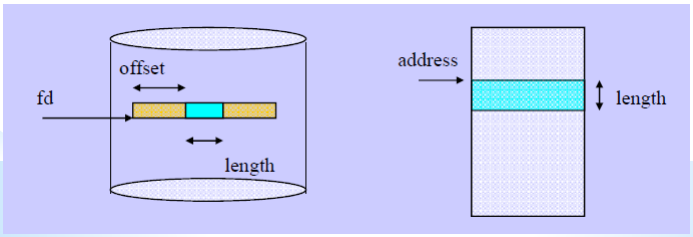
Shared Memory - Operations

- Shared memory object is constructed like a file pathname with an absolute path, and opened with *shm_open()* returning an integer file descriptor
  - *int shm_open(const char *shmname, int oflag, mode_t mode);*
- To set the size of the memory block we use confusing function name *ftruncate()* both to shrink and lengthen the memory
- We use *close()* to remove the shared memory objects and reclaim memory
- We use *mmap()* and *munmap()* to associate and disassociate memory with a process
- Memory protection defines permission for mapping: it must reflect the status of flags used to open the shared memory segment

21

Shared Memory - Use

- To get the created shared memory block into the address space of the process we need to map the descriptor using *mmap()*
- *void * mmap(void *address, size_t length, int protection, int flags, int fd, off_c offset);*
- The system controls the actual assignment and returns the address afterwards
- Once memory address is known, it can be used for all subsequent shared memory allocations



22

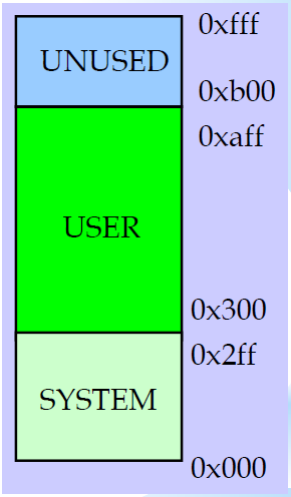
Dynamic Memory Allocation

- Memory is broken into fixed size blocks
- Memory area control block retains information on:
  - starting address
  - overall size
  - allocation array (indicating areas used/free)
- Dynamic allocation is using **heap** data structure to facilitate speed-up the search for free blocks
- In Groups: What is internal fragmentation?
- What is external fragmentation?

23

Memory Map

- Memory is partitioned into two parts:
  - operating system
  - user tasks
- The user tasks may be in:
  - a single partition
  - multiple partitions
- Protection by watching the access address



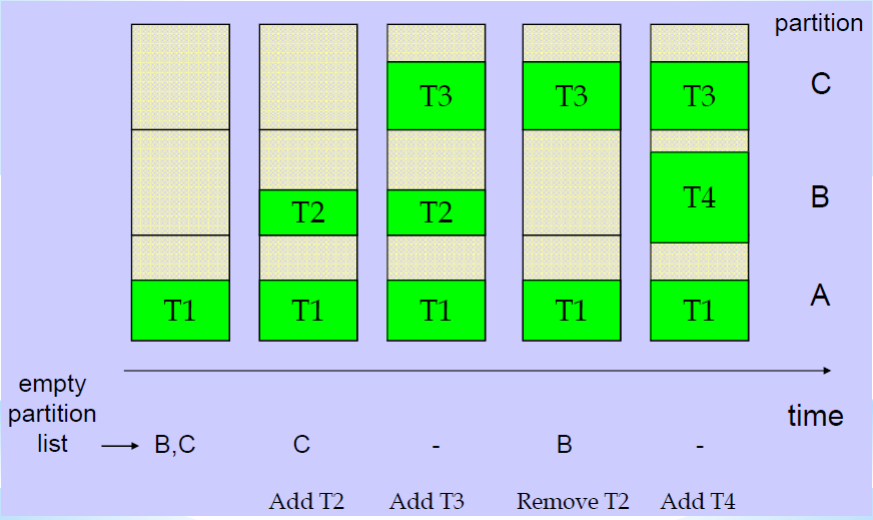
24

Multiple Partition: Fixed Allocation

- Memory divided into fixed size partitions
- When partition is free a task can be loaded
- When task terminates the partition is freed
- The memory manager keeps track of empty partitions
- What kind of system is this good for?

25

Example: Fixed Allocation



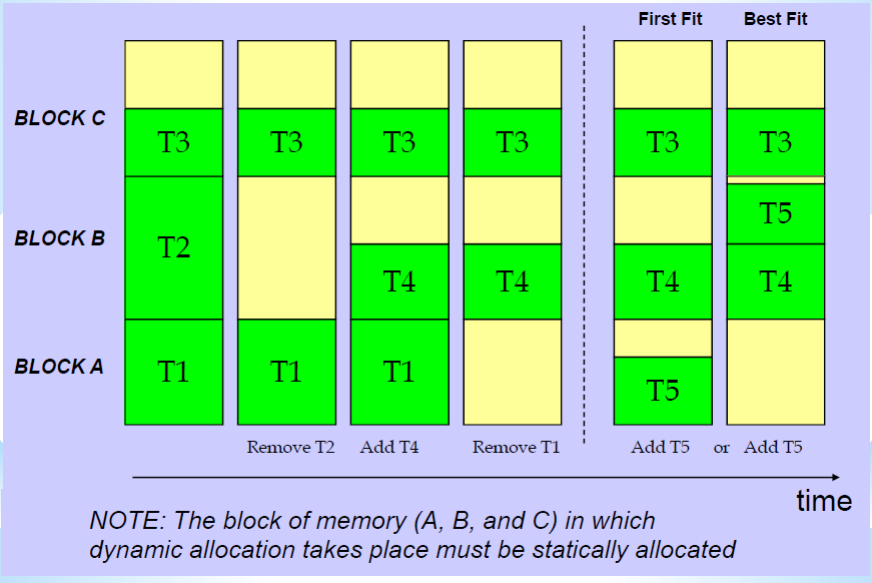
26

Multiple Partition: Dynamic Usage of Statically Allocated Memory

- Initially all memory is available
- New task requests specific memory space
- Memory is assigned in increasing order
- Removal of task creates an empty partition
- The memory manager keeps track of empty partitions (holes) and allocates memory using placement algorithm as:
  - First-fit – the first hole big enough
  - Best-fit – the smallest hole available

27

Example: Dynamic Allocation



NOTE: The block of memory (A, B, and C) in which dynamic allocation takes place must be statically allocated

28

vxWorks Memory Operations

- To dynamically allocate memory: ***void *malloc (nBytes)***
  - Returns a pointer to the allocated memory or *NULL*
  - Uses first-fit algorithm
  - Free memory is stored in a linked list
  - Some (small) overhead for each ***malloc( )***
- To release allocated memory: ***void free (ptr)***
  - Adjacent blocks are merged
- To resize allocated block ***void *realloc (ptr, newSize)***
- To returns size of the largest free block in system memory ***int memFindMax( )***

29

Generic Partition Manager

- VxWorks provides low-level routines to create and manipulate alternate memory pools
- High-level routines like ***malloc( )*** and ***free( )*** call these lower level routines, specifying the system memory pool
- Application may use alternate memory partitions to reduce fragmentation
- Application may use alternate memory partitions to manage memory with different properties

30

Creating and Managing a Memory Partition

will do an example in lab exercise

- ***PART_ID memPartCreate (pPool, size)***
  - ***pPool*** Pointer to memory for this partition
  - ***size*** Size of memory partition in bytes
- Returns a partition ID *(**PART_ID**)*, or *NULL* on error
- System partition management routines call routines listed below, specifying the ***PART_ID*** as ***memSysPartId***

Generic	System Memory Pool
memPartAlloc( )	*malloc( )*
memPartFree( )	*free( )*
memPartShow( )	*memShow( )*
memPartAddToPool( )	*memAddToPool( )*
memPartOptionsSet( )	*memOptionsSet( )*
memPartRealloc( )	*memRealloc( )*
memPartFindMax( )	*memFindMax( )*

31

Memory Management Problems

In groups

- **Memory Leak**
  - when a task continually allocates memory (without proper de-allocation)
- **External Fragmentation**
  - poor allocation and the free memory blocks are too small to accommodate requests
- **Dangling Pointer**
  - a task de-allocates memory but attempts to access it later
- **Poor Locality**
  - the blocks of memory allocated far apart causing performance problems
- **Inflexible Design**
  - different view of memory type of use between memory manager and application
- **Interface Complexity**
  - when data objects are being passed between modules
- **Memory Exhaustion**
  - when a repeated request for service or a growing log depletes memory

32

Summary

- Standard C routines are used for dynamic memory allocation
- To configure the system memory pool:
  - Modify *LOCAL_MEM_SIZE* in ***config.h***
  - Specify *USER_RESERVED_MEM*
  - Call ***memAddToPool( )***
- For fast, deterministic allocation of fixed size buffers,
  - use message queues instead of ***malloc( )***
- Create separate memory partition for off-board memory, or to help reduce fragmentation

33

vxWorks Memory in-class lab

34