1
Introduction

Yu-Lun Huang

1-1
Embedded Systems

Yu-Lun Huang

Real Life Examples

- **Embedded Systems**
  - Presents a class of dedicated computer systems designed for specific purposes
- **Embedded Systems are present in many industries:**
  - Industrial automation
  - Defense
  - Transportation
  - Aerospace
- **Examples:**
  - NASA's Mars Path Finder
  - Lockhead Martin's missile guidance system
  - Ford automobile ..

Definition

- **A general definition of embedded systems is:**
  - Computing systems with tightly coupled **h/w** and **s/w** integration
  - Designed to perform a dedicated function
- **Class I: Truly embedded (i.e., systems within systems)**
- **Case II: Function as standalone systems.**

Truly Embedded

- **Class I: Truly embedded (i.e., systems within systems)**
  - "Embedded" reflects
    - These systems are usually an integral part of a larger system (embedding system)
    - Multiple embedded systems can coexist in an embedding system.
  - Example: Digital set-top box (DST)
    - A/V decoder, an integral part of the DST, is an embedded system
      - IN: A/V decoder accepts a single multimedia stream
      - OUT: A/V decoder produces sound and video frames
    - A/V decoder works with the transport stream decoder (another embedded system)
      - Transport stream decoder de-multiplexes the incoming streams into separate channels and feeds only the selected channel to the A/V decoder

Standalone

- **Case II: Function as standalone systems.**
  - Router is a standalone embedded system
    - Build using a specialized communication processor
    - Memory
    - A number of network access interfaces (ports)
    - Special software (routing algorithms, etc)
  - Router is a standalone embedded system that routes packets from one port to another
- **Focus on the characteristics of embedded systems from many different perspectives to gain a real understanding of what it is and what makes it special.**

Embedded Processor vs. Application Awareness

- **Processors in PC are general-purpose/universal processors.**
  - Complex in design and provide a full scale of features and functionalities
    - Built-in MMU (memory management unit)
      - Provide memory protection & virtual memory for multitasking-capable, general purpose OSs
    - Build-in math co-processor
      - Perform fast floating-point operations
  - Large power consumption, heat production and size

- **What are the advantages and disadvantages of (a) a processor with only fixed-point arithmetic unit and (b) a processor with additional floating-point arithmetic processing unit?**

Embedded Processors (I)

- **Focus on size, power consumption and price**
  - Limited in functionality: good enough for the class of applications but is inadequate for other classes of applications
  - Processor chosen for a PDA
    - Does not have a floating-point co-processor
      - Either not needed, or software emulation is enough
    - Uses 16-bit addressing architecture instead of 32-bit
      - Limited memory storage capacity
    - Have 200MHz CPU speed
      - Applications are interactive & display-intensive (not computation-intensive)
  - **Limited Function: reduced power consumption and long-lasting battery life**
  - **Smaller Size: reduces the overall cost of processor fabrication**

Embedded Processors (II)

- **Focus on performance**
  - Powerful and packed with advanced chip-design technologies
    - Advanced pipeline and parallel processing architecture
  - Designed to satisfy applications with intensive computing requirements not achievable with general-purpose processors
    - Network Processors
      - Developed for network equipment and telecommunication
  - System and application speeds are the main concern

Embedded Processors (III)

- **Focus on all four requirements: performance, size, power consumption and price**
  - Embedded Digital Signal Processor (DSP) used in mobile phone has
    - Specialized arithmetic units
    - Optimized memory design
    - Addressing/bus architectures with multiprocessing capability
      - Perform complex calculations extremely fast in real time
      - Outperform a general-purpose processor running at the same clock speed many times over comes to digital signal processing
      - Reasonably priced to keep the prices of cell phones competitive
      - Battery from which the DSP draws power lasts for hours and hours

Embedded Processors (IV)

- **SoC (System-on-Chip) Processors: attractive for embedded**
  - Comprised of a CPU core with built-in peripheral modules
    - Programmable general-purpose timer
    - Programmable interrupt controller
    - DMA controller
    - Ethernet interfaces (possible)
  - Can be used to build many embedded applications WITHOUT extra external peripheral devices
    - Reduce the overall cost and size of the final product

1-2
Development of Embedded Systems

Co-Design Model

- **Hardware and software for an embedded system are developed in parallel**
  - Software component can take advantage of special hardware features to gain performance
  - Hardware component can simplify module design if functionality can be achieved in software
    - Reduce overall hardware complexity and cost
  - Co-design model emphasized the fundamental characteristics of embedded systems: APPLICATION-SPECIFIC

Cross-Platform Development

- **Platform:**
  - Hardware (processor)
  - Operating System
  - Software Development tools
- **Software for an embedded system is developed on one platform but runs on another.**
  - Host system: where the software is developed
  - Target system: embedded system under development
- **Cross-Compiler**
  - Compiler that runs on one type of processor architecture but produces object code for a different type of processor architecture
    - Host: Intel (IA-32/IA64)
    - Target: ARM/MIPS

Cross-platform

Essential Development Tools

- **Host System offers:**
  - Cross compiler
  - Linker
  - Source-level debugger
- **Target Embedded System offers:**
  - Linker loader
  - Monitor
  - Debug agent
- **Connections between host and target system:**
  - Serial Interface
  - ICE: JTAG (transmit debug info between host debugger & target debug agent)
- **Programs including:**
  - System software
  - Real-time operating system (RTOS)
  - Kernel
  - Application code
- **Steps:**
  - Compiled programs into object code
  - Linked together into an executable image

Development

- **Native development:**
  - Writing applications that execute in the same env as used for development
  - Do not need to be concerned with
    - How an executable image is loaded into memory and
    - How execution control is transferred to the application
- **Cross-platform development:**
  - Need to understand the target system fully:
    - How to store the program image on the target system
    - How and where to load the image during runtime
    - How to develop and debug the system iteratively
    - ..
  - These aspects impact how the code is developed, compiled and **linked**.

Create Image

Basic Concepts of Programming

- **Compiling/linking**
  - Compiling
    - Generate symbol table: symbol name → address
      - global symbols defined in the file being compiled, and
      - external symbols referenced that the linker needs to resolve.
  - Linking
    - In charge of symbol resolution and relocation

Symbol Resolution

- **Symbol resolution is the process in which**
  - Linker goes through each object file and
  - determines which (other) object file(s) the external symbols are defined.
    - Sometimes, linker must process object file(s) multiple times while trying to resolve all of the external symbols.
    - If external symbols are defined in a static library,
      - Linker copies object files from library & writes them to image.
- **For an executable image,**
  - All external symbols must be resolved: each symbol has an absolute memory address
  - Exception: dynamic linking
    - Symbols defined in shared libraries may still contain relative addresses, which are resolved at runtime

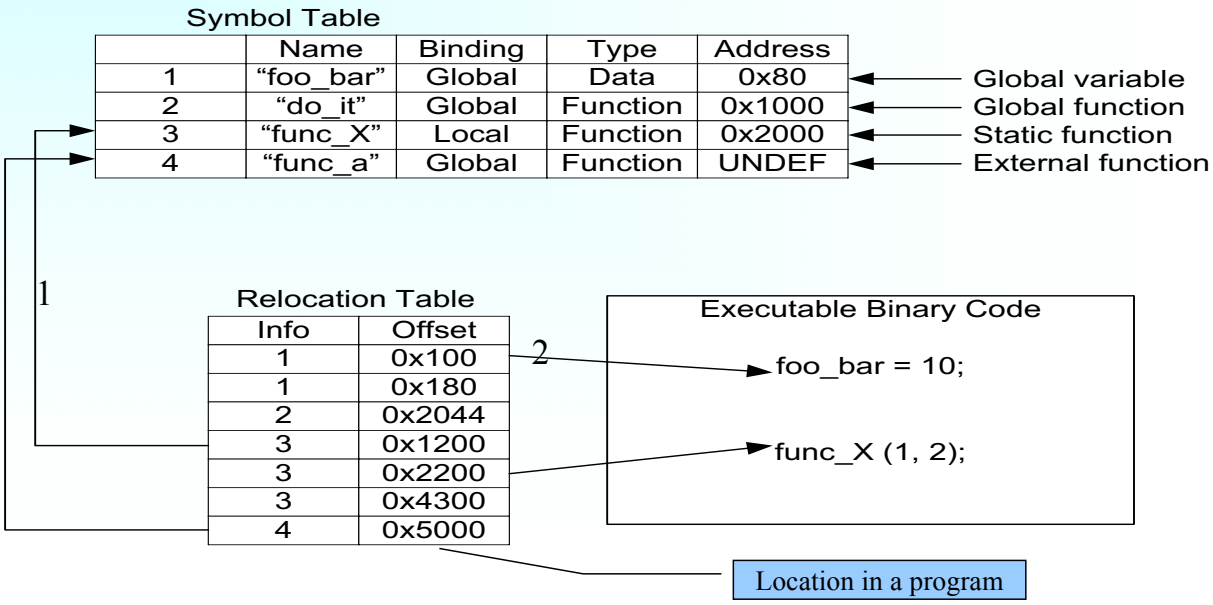
Symbol Relocation

- **Symbol relocation is the process in which**
  - Linker maps a symbol reference to its definition.
  - Linker modifies machine code of the linked object files: code references to the symbols reflect the actual address assigned to these symbols.
  - For many symbols, the relative offsets change after multiple object files are merged.
- **Symbol relocation requires code modification because**
  - Linker adjusts machine code referencing these symbols to reflect their finalized addresses.
- **Relocation table**
  - To tell linker where in the program code to apply the relocation action.
  - Relocation entry contains a reference to the symbol table, thus
    - linker can retrieve actual address of symbol and
    - apply it to program location as specified by relocation entry
  - Relocation table may contain both symbol address & relocation entry information.
    - In this case, there's no reference b/w relocation table & symbol table.

Symbol Relocation (Cont)

- **Relocatable object file may contain unresolved external symbols**
  - Similar to a library, a linker-reproduced relocatable object file is a concatenation of multiple object files. But the file is
    - Partially resolved and
    - Used for further linking w/ other object files to create an executable image or a shared object file.
- **A shared object file has dual purposes:**
  - Used to link with other shared object files or relocatable object modules.
  - Used as an executable image with dynamic linking.

Example



Executable and Linking Format

- **Object file contains**
  - General information about the object file
    - File size, binary code and data size, source file name
  - Machine-architecture-specific binary instructions and data
  - Symbol table and the symbol relocation table
  - Debug information (for debugger uses)
- **Object file format: ways to organize the above information.**
  - Allow tools from different vendors to interoperate with each other
    - compiler, assembler, linker, debugger
  - Two common object file formats (these formats are incompatible):
    - COFF (common object file format)
    - ELF (executable and linking format): supersedes COFF

Software Storage

- **Code for embedded system is stored in ROM & NVRAM**
  - Real-time OS
  - System software
  - Application software
- **ROM**
  - With non-volatile content & without the need for an external power
- **RAM (much faster than ROM)**
  - Requires external power to maintain memory content

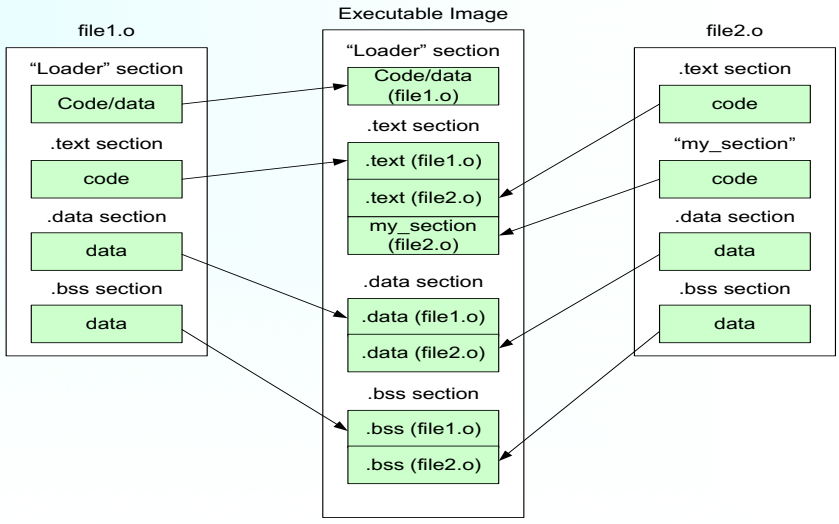
Memory Directive

- Defines types of physical memory on the target system
- Defines address range occupied by each physical memory block

```
MEMORY {
        ROM :       org = 0x0000h,   len = 0x0020h (32 bytes)
        FLASH :     org = 0x0040h,   len = 0x1000h (4096 bytes)
        RAM :       org = 0x10000h, len = 0x10000h (65535 bytes)
        ….
}
```

Section Directive

```
SECTION {
    .text           :
    {
                    my_section
                    *(.text)
    }
    loader          : > FLASH
    GROUP ALIGN (4) :
    {
                    .text,
                    .data:      {}
                    .bss:       {}
    } > RAM
}
```



Why Custom Sections?

- **Module Upgradeability**
  - Upgrade software dynamically when system is running
  - Upgrade involves
    - Downloading new program image over serial line/network
    - Re-programming flash memory
  - Loader capabilities
    - Initial version: transfer image from ROM to RAM
    - Newer version: transfer image from host over serial conn to RAM
  - Custom Section
    - Loader code & data section would be created in custom section
    - Entire section is then programmed into flash for easy upgradeability
- **Memory Size Limitation**
  - Target system has different types of physical memory with limited sizes
  - If SDRAM is not large enough to fit all, but lots of DRAM is available:
    - Divide program into multiple sections
      - Some allocated in SDRAM: frequently used functions/tables
      - Rest mapped into DRAM: remaining code and data

Why Custom Sections? (cont)

- **Data Protection**
  - Programs usually have various types of constants
    - Integer constants and string constants
  - Some constants are kept in ROM to avoid accidental modification
    - Put these constants into a special data section, which is allocated into the ROM.

1-3
Embedded OS

EOS

- **EOS kernel is developed instead as a micro-kernel**
  - Reduce essential kernel services into a small set
  - Provide a framework: optional kernel services can be implemented as modules
    - Modules can be placed outside the kernel
    - Some modules are part of special server tasks
  - Extend the kernel by adding additional services or to modify existing services without affecting users
  - Each embedded application requires a specific set of services with respect to its characteristics
    - Combination can be quite different from application to application
- **Micro-kernel provides core services, including**
  - Task-related services
  - Scheduler service
  - Synchronization primitives

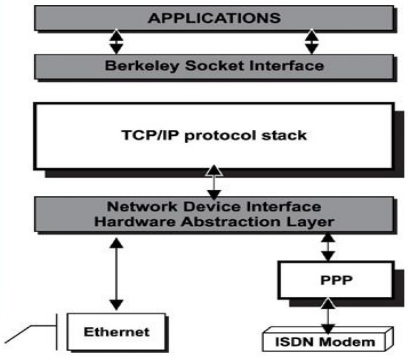
RTOS Overview

Building Blocks

- **TCP/IP protocol stack**
- **File system component**
- **Remote procedure call component**
- **Command shell**
- **Target debut agent**
- **Other components**

TCP/IP Protocol Stack

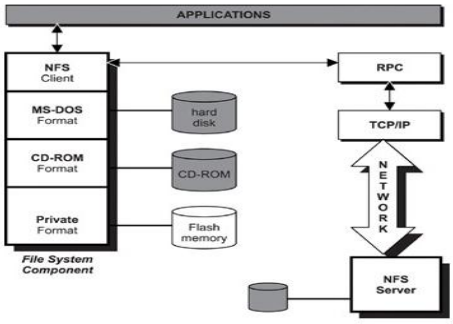
- **TCP/IP protocol stack provides transport services to both higher layer, well-known protocols, including**
  - Simple Network Management Protocol (SNMP)
  - Network File System (NFS)
  - Telnet
  - User-defined protocols



- **Transport service can be either**
  - reliable connection-oriented service over TCP
  - unreliable connectionless service over UDP
- **TCP/IP protocol stack can operate over**
  - Ethernet, Frame Relay, ATM, ISDN networks, …
    - Using different frame encapsulation protocols, including PPP
  - Commonly, transport services offered through standard Berkeley socket interfaces.

File System

- **File system component provides efficient access to both local and network mass storage devices**
  - CD-ROM, tape, floppy disk, hard disk, and flash memory devices...
- **File system component structures storage device into supported formats for reading/writing to storage device**
  - CD-ROMs are formatted and managed according to ISO 9660 standard file system specifications
  - Floppy disks and hard disks are formatted and managed according to MS-DOS FAT file system conventions and specifications
  - NFS allows local applications to access files on remote systems as an NFS client
    - Files located on an NFS server are treated exactly as though they were on a local disk
      - NFS is a protocol, not a file system format, local applications can access any format files supported by the NFS server



Component Configuration

- **Since memory in embedded system is limited, only key components are selected into final application image**
- **How to configure a service component into an embedded application?**
  - System configuration files: selection and configuration of service components are accomplished through these files
- **Example of component header file**

```
#define INCLUDE_TCPIP 1
#define INCLUDE_FILE_SYS 0
#define INCLUDE_SHELL 1
#define INCLUDE_DBG_AGENT 1
```

  - Target image includes TCP/IP protocol stack, command shell, and debug agent

2
Multitasking

2-1
Task & Scheduler

Defining a Task

- **Task: an independent thread of execution**
  - Developers decompose applications into multiple concurrent tasks: optimize the handling of inputs/outputs within set time constraints
- **Task is Schedulable**
  - Can compete w/ other concurrent tasks for processor execution time
- **Task Creation**
  - Defined by distinct set of parameters and supporting data structures
    - Task Objects:
      - Name
      - Priority
      - Stack (size)
      - Task routine
      - Unique ID
      - Task control block (TCB)

What is Task?

Task Creation

- **Kernel creates system tasks**
- **Kernel starts**
  - **Initialization or startup task**: initializes, creates and starts system tasks
  - **Idle task**: uses up processor idle cycles when no other activity is present
  - **Logging task**: logs system messages
  - **Exception-handling task**: handles exceptions
  - **Debug agent task**: allows debugging with a host debugger
- **Kernel allocates priority from reserved priority levels**
  - Reserved priority levels: used internally by RTOS & its system tasks
  - Applications should avoid using these priority levels
    - Running at such level may affect system performance/behavior
  - Reserved priorities are not enforced in most RTOSes
  - Kernel needs its system tasks and their reserved priority levels to operate.
  - These priorities should not be modified
- **Kernel jumps to a predefined entry point: application**
  - Then, developer can initialize and create other application tasks

Idle Task

- **Created at kernel startup**
- **Set to the lowest priority:**
  - executes in an endless loop and
  - runs when either no other task can run or when no other tasks exist,
  - Purpose: using idle processor cycle
- **Idle task is necessary because**
  - Processor executes instruction to which program counter register points
  - Unless the processor can be suspended, the program counter must still point to valid instructions even when no tasks exist in the system or when no tasks can run.
  - Idle task ensures the processor program counter is always valid
- **Special Requirement: user-configured routine vs. idle task**
  - Power conservation: make system suspend after a period of idle time

Task States

- **States: ready, running and blocked**
- **Preemptive-scheduling kernels contain following states:**
  - Ready: task is ready to run but cannot (higher priority task is executing)
  - Blocked, if:
    - Task has requested a resource that is not available,
    - Task has requested to wait until some event occurs or
    - Task has delayed itself for some duration
  - Running: task is the highest priority task and is running
- **Note: some kernels define more states**
  - Suspended, pended and delayed
    - Pended/delayed are sub-states of blocked state
      - Pended: waiting for a resource that needs to be freed
      - Delayed: waiting for a timing delay to end.
    - Suspended: for debugging purposes

Task States (cont)

- **Kernel scheduler determines which tasks need to change states and then makes those changes**
- **Upon changing states,**
  - No context switching occurs: state of highest priority task is unaffected.
  - Context switching occurs:
    - Former highest priority task gets blocked or
    - Former highest priority task is no longer the highest priority
    - Former highest priority task is put into blocked or ready state
    - New highest priority task starts to execute

Ready State

- **Upon creation, kernel puts the task into READY state**
  - Actively competes with all other ready tasks for processor's execution time.
  - Cannot move directly to the BLOCKED state: *need make blocking call*
    - Immediately run to completion and put the task in the blocked state.
- **Kernel scheduler uses the priority of each task to determine which task to move to the RUNNING state.**

Running State

- **Single-processor system: only one task can run at a time**
  - Task moves to RUNNING state
  - Processor loads its registers with task's context
  - Processor then execute task's instructions and manipulate the stack
- **RUNNING ->READY: preempted by higher priority task**
- **RUNNING->BLOCKED, if a task:**
  - Makes a call that requests an unavailable resource
  - Makes a call that requests to wait for an event to occur
  - Makes a call to delay the task for some duration

Blocked State

- **In real-time systems, BLOCKED is extremely important:**
  - Lower priority tasks cannot run without blocked state
  - CPU starvation can result if higher priority tasks aren't designed to block
- **Task can only move to BLOCKED by a blocking call**
- **Blocked task still blocked until *blocking* condition is met**
  - blocking here means unblocking
- **Blocking Condition:**
  - Semaphore token for which a task is waiting is released,
  - Message, on which the task is waiting, arrives in message queue
  - Time delay imposed on the task expires
- **When unblocked, a task**
  - Move from BLOCKED to READY: if not the highest priority task
  - Move from BLOCKED to RUNNING: if the highest priority task
    - Preempts the currently running task

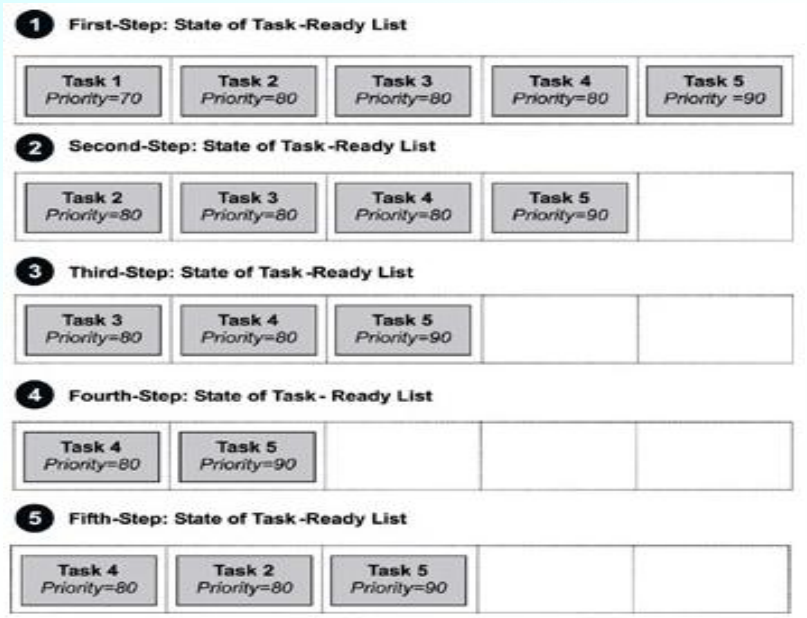
Kernel Scheduler

- **Single task per priority level**
  - Highest priority READY task runs next.
  - Kernel limits #tasks in an application to #priority levels
- **Multiple tasks per priority level**
  - Many tasks can be run in an application
  - Scheduling algorithm involves a task-ready list.
  - Some kernels maintain a separate task-ready list for each priority level; others have one combined list.

Example of Task Scheduler

- **Assumption**
  - Single-processor system
  - Priority-based preemptive scheduling algorithm (lowest: 255, highest: 0)
- **Description**
  - Task 1(70), 2(80), 3(80), 4(80), and 5(90) are ready to run
  - Kernel queues them in task-ready list: by priority
- **Steps**
  - Task 1, 2, 3, 4, and 5 are ready to run: waiting in task-ready list
  - Kernel removes Task 1 from ready list and moves it to RUNNING state
  - Task1 makes a blocking call
    - Task1 moves to BLOCKED state.
    - Kernel removes Task 2 from list and moves it to RUNNING state
  - Task2 makes a blocking call
    - Task 2 moves to BLOCKED state.
    - Kernel removes Task 3 from list and moves it to RUNNING state
  - Task 3 frees the resource that Task 2 requested.
    - Kernel moves Task 2 to READY and insert it to the end of the list
    - Task 3 continues as the currently running task.

Task-ready List



Task Scheduling

- **Manual scheduling**
  - Control when a task moves to a different state by kernel APIs
- **Operations for manual scheduling**
  - Suspend: suspends a task
  - Resume: resumes a task
  - Delay: delays a task
  - Restart: restarts a task
  - Get priority: get the current task's priority
  - Set priority: dynamically sets a task's priority
  - Preemption lock: locks out higher priority task from preempting the current task
  - Preemption unlock: unlocks a preemption lock
- **Why manual scheduling?**
  - Debugging
  - Make lower priority tasks be able to run

Task Structure

- **Task are structured in one of two ways:**
  - Run to completion
  - Endless loop
- **Run-to-completion: useful for initialization and startup**
  - Run once, typically
- **Endless-loop: do the majority of the work in the application**
  - Handling inputs and outputs
  - Run many times while the system is powered on

Run-to-Completion Tasks

- **Application-level initialization task**

```
RunToCompletionTask()
{
   Initialize application
   Create 'endless loop tasks'
   Create kernel objects
   Delete or suspend this task
}
```

- **Normally has a higher priority then tasks it create**
  - Initialization work is not preempted

Endless-Loop Tasks

- **One or more blocking calls within the body of the loop**
  - Result in the blocking of this endless-loop task, allowing lower priority tasks to run

```
EndlessLoopTask()
{
   Initialization code
   Loop Forever
   {
      Main job here..
   }
}
```

2-2
Linux Process

Linux Process

- **task_struct is used to present each process**
  - process and task are interchangeable in Linux
- **task_vector is a pointer array, which keeps pointer to each task_struct of the corresponding process**
- **On creation, kernel allocates a task_struct for the newly created process and puts its address to the task_vector**
- task_struct **contains**
  - State
  - Scheduling information
  - Identifier
  - IPC (inter-process communication)
  - Link
  - Times and timers
  - File system
  - Virtual memory
  - Processor specific context

Task_struct: State

- **Running**
- **Waiting**
  - Process is waiting for an event or a resource
  - There are two types of waiting states: interruptible, uninterruptible
- **Stopped**
- **Zombie**
  - For some reason, there is a pointer of a terminated task in task_vector

Task_struct: Others

- **Scheduling information**
  - Used to determine the next process to be executed
- **Identifier**
  - Used to control the process in accessing system files and resources
  - Not the index in task_vector
- **IPC**
  - Signal, pipe, semaphore, share memory, message queue
- **Link**
  - Used to reference to process' parent, brother and child processes
  - pstree can be used to check the relationship of the running processes
- **Times and timer**
- **File system**
  - One file descriptor and two VFS inode (one for root directory of process and the other for current directory)
- **Virtual memory**
  - Mapping between virtual memory and physical memory
- **Context**
  - Keeps status of registers, stacks, etc

2-3
Linux Thread

Designing Threaded Programs

- In order for a program to take advantage of threads, it must be able to be organized into discrete, independent tasks which can execute concurrently
- For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading

Designing Threaded Programs

- **Tasks that may be suitable for threading include tasks that:**
  - Block for potentially long waits
  - Use many CPU cycles
  - Must respond to asynchronous events
  - Are of lesser or greater importance than other tasks
  - Are able to be performed in parallel with other tasks
- **Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness**
  - When in doubt, assume that they are not thread-safe until proven
- **Several common models for threaded programs exist:**
  - *Manager/worker:* a single thread, the *manager* assigns work to other threads, the *workers*. Typically, the manager handles all input and parcels out work to the other tasks.
  - *Pipeline:* a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line best describes this model.
  - *Peer:* similar to the manager/worker model, but after the main thread creates other threads, it participates in the work

Process vs. Thread

- **All threads within a process share the same address space**
  - Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication
- **Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:**
  - Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
  - Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
  - Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests


Multithreaded programming
Theory
Actual
VIA FUNNYMEME.COM

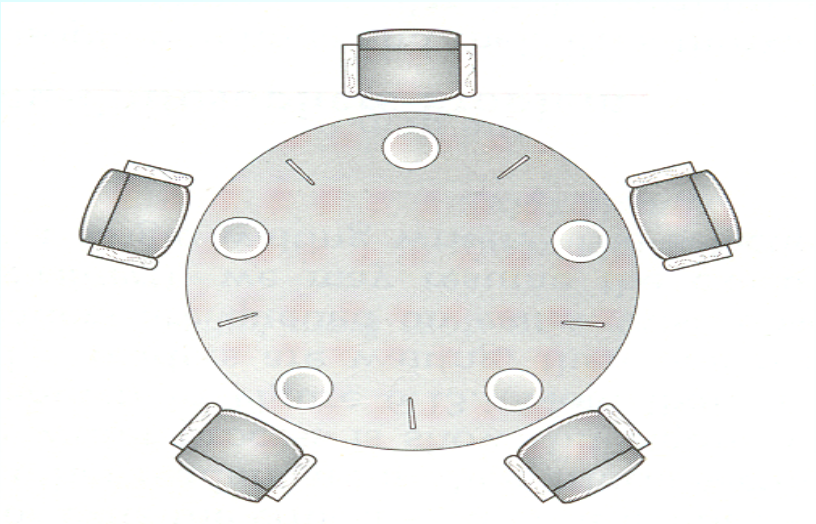
3
Communication & Synchronization

3-1
Inter-Process Communication (IPC)

Inter-Process Communication

- Semaphore/Mutex
- Message Queue
- Pipe
- Shared Memory
- …

3-1-1 Dining Philosophers



Defining Semaphores

- **Semaphore is a kernel object**
  - One or more threads of execution can acquire or release
  - For synchronization or mutual exclusion
    - Key to carry out some operation or to access resources
- **When creating semaphore,**
  - Kernel assigns to it
    - Semaphore control block (SCB)
    - Unique ID
    - Value (binary or count)
    - Task-waiting list
- **Types of Semaphores**
  - Binary semaphores
  - Counting semaphores
  - Mutual-exclusion (mutex) semaphores

Binary Semaphores

- **Binary semaphore can have a value of either 0 or 1.**
  - 0: semaphore is unavailable (empty)
  - 1: semaphore is available (full)
    - Binary semaphore can be initialized to either 0 or 1
- **Binary semaphores are treated as global resources**
  - Shared among all tasks that need them
  - Any task can release it (even if the task did not initially acquire it)
- **State diagram of a binary semaphore**

Counting Semaphores

- **Uses a count to allow multiple acquisition or release**
  - Count = 0: counting semaphore moves from available → unavailable
  - Unavailable → available: release at least one token
    - Assign the semaphore a count: denotes #tokens it has initially
      - 0: created in unavailable state; > 0: created in available state
- **Counting semaphores are treated as global resources**
  - Shared among all tasks that need them
  - Any task can release it (even if the task did not initially acquire it)
- **Bounded vs. Unbounded Count**
  - Unbounded: maximum value that can be held by the count's data type
    - Unsigned integer vs. unsigned long value
  - Bounded: Maximum count
- **State Diagram**

Mutex Semaphores

- **Special binary semaphore**
  - Supports ownership, recursive access, task deletion safety and protocols for avoiding mutual exclusion problems
- **Locked (0)/Unlocked (1) vs. acquire/release**
  - Initially, mutex is created in unlocked state: can be acquired by a task
  - After being acquired, mutex moves to locked state
  - After releasing, mutex returns to unlocked state
- **Depending on implementation, mutex can support**
  - Features not found in binary or counting semaphores
    - Ownership, recursive locking, task deletion safety, priority inversion avoidance protocols
- **State diagram**

Mutex Ownership

- **Ownership**
  - Gained when task first locks mutex by acquiring it
  - Lost: when task unlocks mutex by releasing it
- **Task owns the mutex: Not possible for any other task to lock/unlock that mutex**
  - Binary semaphore: can be released by any task, even if it did not acquire the semaphore

Recursive Locking

- **Recursive locking: allows task (mutex owner) to multiple acquire it in the locked state**
  - Recursive Mutex
    - Be useful when task (requires exclusive access to a shared resource) calls one or more routines (also require access to the same resource)
    - Allow nested attempts to lock the mutex: Rather than cause deadlock
      - Deadlock: two or more tasks are blocked and are waiting on mutually locked resources
  - When a recursive mutex is first locked, kernel registers the task as owner
  - On successive attempts, kernel uses internal lock count for #acquiring
  - To unlock the mutex, it must be released the same number of times
  - Lock count tracks: 1. states of mutex (0 or 1), 2. #recursive locks
    - What's different b/w count for mutex & counting semaphore?
      - Acquire/Release: owner task vs. any task
      - Unbounded vs. bounded: mutex count is always unbounded

Task Deletion Safety

- **Premature task deletion is avoided by using task deletion locks when a task locks and unlocks a mutex**
  - Task can not be deleted if it owns a mutex
  - Task deletion safety option is enabled when creating the mutex

Priority Inversion Avoidance

- **Occurs when**
  - Higher priority task is blocked/waiting for a resource being used by a lower priority task
  - Lower priority task has been preempted by medium priority task
  - Result: higher priority task's priority level has effectively been inverted to the lower priority task's level

- **Protocols used for avoiding priority inversion:**
  - Priority inheritance protocol
    - Priority of lower priority task is raised to that of higher priority task
    - Priority of raised task is lowered to original value a/f releasing mutex
  - Ceiling priority protocol
    - Priority of task that acquires mutex is set to the highest priority
    - When mutex is released, the priority is lowered to its original value

Common Semaphore Use

- **Waiting-and-signal synchronization**
- **Multiple-task wait-and signal synchronization**
- **Single shared-resource-access synchronization**
- **Recursive shared-resource-access synchronization**
- **Multiple shared-resource-access synchronization**

Wait-and-signal

- **Two tasks comm. for synchronization w/o exchanging data**
  - Use binary semaphore to coordinate the transfer of execution control
    - Binary semaphore is initially unavailable (0)
    - tWaitTask has higher priority and runs first
    - Task makes a request to acquire the semaphore but is blocked
    - tSignalTask runs and releases the binary semaphore (1)
    - tWaitTask (higher priority) is unblcoked
    - tWaitTask preempts tSignalTask as soon as semaphore is released
    - tWaitTask starts to execute
- **Pseudo code**

```
tWaitTask() {
    :
Acquire binary semaphore token
    :
}
```

```
tSignalTask() {
    :
Release binary semaphore token
    :
}
```

Multiple-Task Wait-and-Signal

- **When coordinating sync of more than two tasks**
  - Use flush operation on the task-waiting list (twl) of binary semaphore
    - Binary semaphore is initially unavailable (0)
    - tWaitTasks 1, 2, 3 try to acquire semaphore (0) & are blocked
    - tSignalTask has a chance to do flush & unblock 3 tWaitTasks
    - As soon as the semaphore is released, one of the tWaitTasks preempts tSignalTask and starts execution

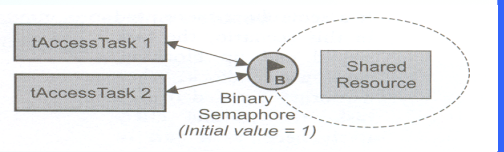
- **Pseudo code**

```
tWaitTask() {
Process specific job
Acquire binary semaphore token
    :
}
```

```
tSignalTask() {
Process something
Flush binary semaphore task-waiting list
    :
}
```

Single Shared-Resource-Access

- **Goal: access shared resources**
  - memory location, data structure, I/O devices
- **Create binary semaphore**
  - In AVAILABLE state (1), initially
- **Use binary semaphore to protect shared resource**
- **Task needs acquire token before r/w to the resource**
- **Danger**
  - Any task can accidentally release the semaphore
  - Use mutex to solve the problem
- **Pseudo code**




```
tAccessTask() {
    :
Acquire binary semaphore token
Read/Write to shared resource
Release binary semaphore token
    :
}
```

Recursive Shared-Resource-Access

- **Goal: access shared resource recursively**
  - tAccessTask calls Routine A, and Routine A calls Routine B
  - All need access to the same shared resource
  - Deadlock: task would end up blocking
    - Routine A is part of tAccessTask
    - Routine A acquires the semaphore
    - tAccessTask tries to acquire the same semaphore
    - tAccessTask ends up blocking while waiting for the semaphore it already has
  - Use recursive mutex to solve the problem
    - After tAccessTask locks the mutex, task owns it
    - Additional attempts from the task or from routines to lock the mutex succeed
      - Routines attempt to lock the mutex succeed without blocking
- **Pseudo Code**

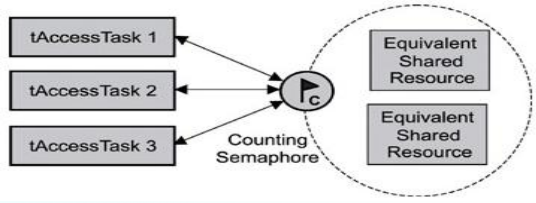
```
tAccessTask() {
    :
Acquire mutex
Read/Write to shared resource
Call Routine A
Release mutex
}
```

```
Routine A() {
    :
Acquire mutex
Read/Write to shared resource
Call Routine B
Release mutex
}
```

```
Routine B() {
    :
Acquire mutex
Read/Write to shared resource
Release mutex
    :
}
```

Multiple Shared-Resource-Access

- **Multiple equivalent shared resources are used**
  - Counting semaphore comes in handy: set to #eq shared resources
    - Does not work if shared resources are **NOT** equivalent
- **Example**
  - #eq shared resources: 2
  - #access tasks: 3
    - 3rd task ends up blocking till one of previous 2 tasks releases a token
- **Problem:**
  - If a task releases a semaphore it did not acquire
  - Mutexes can provide built-in protection




```
tAccessTask() {
    :
Acquire a counting semaphore token
Read/Write to shared resource
Release counting semaphore token
}
```

```
tAccessTask() {
    :
Acquire 1st mutex in a non-blocking way
  If failed, acquire 2nd mutex in non-blocking way
Read/Write to shared resource
Release the acquired mutex
}
```

3-1-2 Message Queues

Defining Message Queues

- **Message queue is a buffer-like object**
  - Tasks/ISRs can send/recv mesg to comm/sync through message queue
- **Message queue is like a pipeline**
  - Temporarily holds mesg from sender until receiver is ready to read
- **When created, it is assigned**
  - Associated queue control block (QCB)
  - Message queue name
  - Unique ID
  - Memory buffers
  - Queue length
  - Maximum message length
  - One or more task-waiting lists,

Defining Message Queues (cont)

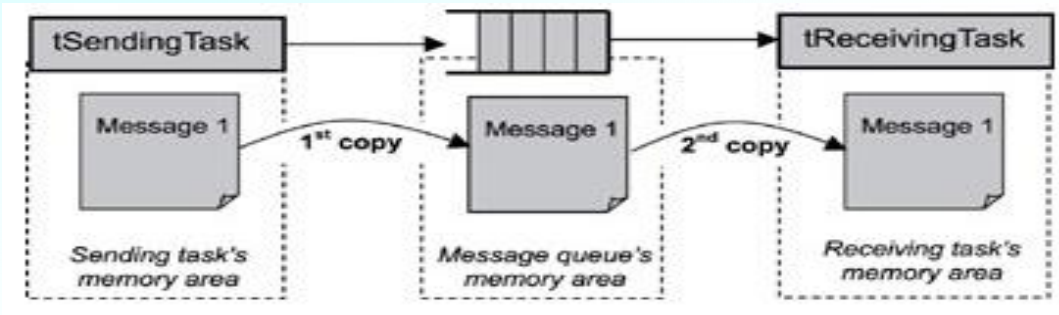
- **Message queue consists of many elements: hold a single message**
  - Total number of elements in the queue is the *total length of the queue* .
  - Developer specified the queue length when created.
- **A message queue has two associated task-waiting lists**
  - Receiving list: consists of tasks that wait on the queue when it is empty.
  - Sending list: consists of tasks that wait on the queue when it is full.

Message Queue States

- **States: Empty, not-empty, full**
- **Empty:**
  - If task sends msg to queue, msg is delivered directly to blocked task
    - Remove task from task-waiting list: to READY/RUNNING.
    - Queue remains empty because it successfully delivered msg.
- **Not Empty:**
  - If msg is sent to queue and no tasks are waiting in queue's task-waiting list, state becomes not empty.
- **Full:**
  - Queue eventually fills up until it has exhausted its free space
    - #messages in the queue = queue's length
- **Receiving List:**
  - Task attempts to receive messages from this message queue (empty)
  - Task blocks & is held on task-waiting list in FIFO/priority-based order
- **Sending List:**
  - When FULL, task sending msgs to queue will be failed unless other task first requests a msg and freeing a queue element

Memory Use

- **Sending/Receiving Messages**



- **When a task attempts to send a message to a full queue,**
  - Sending function of some implementation returns error code to task
  - Other implementations allow such a task to block, moving the blocked task into the sending task-waiting list

Message Queue Content

- **Message Queue can be used to send/receive a variety of data:**
  - Temperature value from a sensor,
  - Bitmap to draw on a display,
  - Text message to print to an LCD,
  - Keyboard event, and
  - Data packet to send over the network.
- **Messages longer than Maximum message length**
  - Send a pointer to the data, rather than the data itself
    - This also improves both performance and memory utilization
- **On sending msgs to another task, msg normally is copied twice**
  - 1st: when msg is sent from sending task's mem area to queue's mem area
  - 2nd: when msg is copied from queue's mem area to receiving task's mem area.
- **An exception to this situation is**
  - Receiving task is already blocked waiting at the message queue
    - Depending on implementation, msg might be copied just once:
      - from sending task's memory area to receiving task's memory area,
      - bypassing copy to queue's memory area

Message Queue Storage

- **Diff kernels store message queues in diff memory locations**
  - System pool or private buffers
- **System Pools**
  - Msgs of all queues are stored in ONE large shared memory area
  - Good if making sure all queues will never be filled up at the same time
    - This approach saves on memory use, but
    - Queue with large msgs may use most of the pooled memory, not leaving enough memory for other queues. Results in:
      - » 1. Other queue (not full) starts rejecting messages sent to it
      - » 2. A full queue continues to accept more messages
- **Private Buffers**
  - Requires enough reserved memory area for full capacity of every queue
  - This approach clearly uses up more memory, but ensures that
    - Messages do not get overwritten
    - Room is available for all messages, resulting in better reliability

Sending Messages

- **Kernel fills queue from head to tail in FIFO order**
  - Place new msg at the end of queue
- **Many implementations allow urgent msgs to go straight to the head**
  - All msgs are urgent: all go to the head of queue: LIFO

Sending Messages (cont)

- **Block**
  - Block task if attempt to send msgs to a queue that is full
    - Block forever or for a specified timeout
      - Blocked task is placed in queue's task-waiting list
        - » in either FIFO or
        - » priority-based order
    - Block forever: blocks until a queue element becomes free
    - Block for a specified timeout: unblocked if either
      - A queue element becomes free or
      - Timeout expires (returns error)

Receiving Messages

- **Task can receive messages with different blocking policies**
  - Not blocking,
  - Blocking with a timeout, or
  - Blocking forever
- **Blocking occurs due to empty queue, and receiving tasks wait in either FIFO or priority-based order**
- **For a queue to become full, either**
  - Receiving task list must be empty or
  - Rate of sending must be greater than the rate of receiving
- **Only when queue is full does sending list start to fill**
- **For receiving list to start to fill, queue must be empty**
- **Msgs can be read from the head of queue in two ways:**
  - destructive read, and non-destructive read.
- **Destructive read**
  - On receiving, task permanently removes msg from queue buffer
- **Non-destructive read**
  - Receiving task peeks at the msg at the head of queue without removing it
  - Not all kernel implementations support the non-destructive read.

Obtaining Message Queue Info

- **Different kernels allow developers to obtain different types of information about a message queue, including**
  - Message queue ID,
  - Queuing order used for blocked tasks (FIFO or priority-based), and
  - #messages queued
  - Full list of messages that have been queued up.
- **Information is dynamic and might have changed by the time it's viewed**
- **These types of calls should only be used for debugging**

Typical Use

- **Typical ways to use message queues :**
  - non-interlocked, one-way data communication,
  - interlocked, one-way data communication,
  - interlocked, two-way data communication, and
  - broadcast communication

Non-Interlocked, One-Way Data Communication

- **tSourceTask only sends msg (no ACK from tSinkTask): loosely coupled**
- **tSinkTask is set to a higher priority:**
  - tSinkTask runs first until it blocks on an empty message queue
  - When tSourceTask sends msg to queue, tSinkTask recvs msg & starts to exe
- **tSinkTask is set to a lower priority:**
  - tSourceTask fills the message queue with messages
  - Finally, tSourceTask can be made to block when sending msg to a full queue
  - tSinkTask wake up and start taking messages out of the message queue.
- **ISRs typically use non-interlocked, one-way comm**
- **NOTE: ISRs must send msgs to queue in a non-blocking way**
  - If queue is full, any additional messages that the ISR sends to queue are lost
- **Pseudo code**

```
tSourceTask (){
    :
  Send message to message queue
    :
}
```

```
tSinkTask () {
    :
  Receive message from message queue
    :
}
```

Interlocked, One-Way Data Communication

- **If ACK on receiving msg is needed: interlocked comm.**
  - Sending task sends a msg and waits to see if it is received
  - Be useful for reliable comm or task sync
- **tSourceTask and tSinkTask use**
  - binary semaphore (0): sync object which acts as a simple ACK
  - message queue with a length of 1 (also called a mailbox)
- **Steps**
  - tSourceTask sends msg to queue and blocks on binary semaphore
  - tSinkTask receives the message and increments the binary semaphore
  - Semaphore (1) wakes up tSourceTask
  - tSourceTask, which executes and posts another msg into queue, blocking again afterward on the binary semaphore
- **Pseudo code**

```
tSourceTask (){
    :
  Send message to message queue
  Acquire binary semaphore token
}
```

```
tSinkTask () {
    :
  Receive message from message queue
  Release binary semaphore token
}
```

Interlocked, Two-Way Data Communication

- **Data flow bidirectionally b/w tasks: interlocked, two-way data comm (full-duplex or tightly coupled comm)**
  - Be useful when designing a client/server-based system
  - Two message queues are required
  - tServerTask is typically set to a higher priority to quickly fulfill requests
  - If multiple clients, all clients can use request queue to post requests
  - tServerTask uses separate server queue to fulfill the requests
- **Steps**
  - tClientTask sends a request to tServerTask via a message queue
  - tServer-Task fulfills that request by sending a msg back to tClientTask
- **Pseudo Code**

```
tClientTask (){
    :
  Send message to request queue
  Wait message from server queue
}
```

```
tServerTask () {
    :
  Receive message from request queue
  Send message to server queue
}
```

Broadcast Communication

- **Broadcast a copy of the same msg to multiple tasks**
- **Msg broadcasting is a one-to-many-task relationship**
- **Steps**
  - tBroadcastTask sends msg on which multiple tSink-Task are waiting.
  - tSinkTasks all made calls to block on broadcast queue: wait for msg
  - tBroadcastTask exes/sends 1 msg to queue: all tasks exiting blocked state
- **Note**
  - Not all implementations support broadcasting facility
- **Pseudo Code**

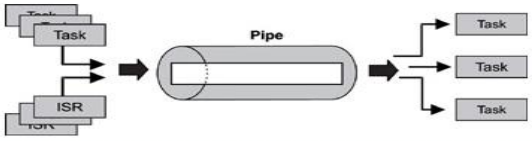
```
tBroadcast (){
    :
  Send broadcast message to queue
    :
}
```

```
tSinkTask () {
    :
  Receive message from queue
    :
}
```

3-1-3 Pipe

Pipe

- **Provide UNSTRUCTURED data exchange and facilitate synchronization among tasks.**
  - Traditionally, pipe is unidirectional
    - Two descriptors: for reading and writing
    - Data is written via one descriptor and read via the other
    - Data remains in the pipe as an unstructured byte stream
    - Data is read from the pipe in FIFO order
- **Provide a simple data flow facility:**
  - Reader becomes blocked when the pipe is empty,
  - Writer becomes blocked when the pipe is full.
  - Typically, a pipe is used to exchange data between a data-producing task and a data-consuming task
  - Multiple writers vs. Multiple readers



Pipe vs. Message Queue

- **Similarity**
  - Data exchange/communication
  - Task synchronization
- **Difference**
  - Pipe does not store multiple messages
    - Instead, data that it stores is not structured, but consists of a stream of bytes.
  - Pipe does not support priority
    - Data in a pipe cannot be prioritized
    - Data flow is strictly first-in, first-out FIFO
  - Pipe supports select operation, and message queue does not

Pipe Control Block (PCB)

- **Generally, a pipe control block contains**
  - Kernel-allocated data buffer for the pipe's input and output operation
    - Size of this buffer is maintained in the control block and is fixed when the pipe is created; it cannot be altered at run time.
  - Current data byte count: amount of readable data in the pipe
  - Current input position indicator: where the next write begins in buffer
  - Current output position indicator: where the next read in buffer
- **Two task-waiting lists are associated with each pipe**
  - One keeps track of tasks waiting to write into the pipe when full
  - The other keeps track of tasks waiting to read from the pipe when empty

Pipe States

- **Empty, Not Empty, Full**
  - Each state corresponds to the data transfer state b/w reader and writer

Select

- **SELECT: allows a task to block and wait for a specified condition to occur on one or more pipes**
  - Wait condition can be waiting for data to become available or waiting for data to be emptied from the pipe(s)
- **Example**
  - A task is waiting to read from two pipes and write to a third
    - SELECT() returns when data is available on the top of either 2 pipes
    - Same SELECT() returns when space for writing becomes available on the bottom pipe
    - A task reading from multiple pipes can perform SELECT() on pipes, and SELECT() returns when any one of them has data available
    - A task writing to multiple pipes can perform SELECT() on pipes, and SELECT() returns when space is available on any one of them.

Typical Uses of Pipes

- **Common use**
  - Task-to-task or ISR-to-task data transfer
  - Inter-task synchronization
    - Can be asynchronous for both tasks by using SELECT()
- **Example**
  - Task A and B open two pipes for inter-task communication
    - First pipe is opened for data transfer from task A to task B
    - Second is opened for ACK (another data transfer) from task B to A
    - Both tasks issue SELECT() on the pipes
      - Task A can wait asyncly for data pipe to become writeable
        - » Task B has read some data from the pipe
      - Task A can wait asyncly for ACK from task B on the other pipe
      - Task B can wait asyncly for data arrival on data pipe
      - Task B can wait for the other pipe to become writeable before sending ACK

3-1-4 Shared Memory

Shared Memory in Linux

- **Shared memory allows multiple processes to share virtual memory space.**
- **This is the fastest but not necessarily the easiest (synchronization-wise) way for processes to communicate with one another.**
- **In general, one process creates or allocates the shared memory segment.**
  - The size and access permissions for the segment are set when it is created.
- **The process then attaches the shared segment, causing it to be mapped into its current data space.**
- **For each process involved, the mapped memory appears to be no different from any other of its memory addresses.**

Creating a Shared Memory Segment

- **The shmget system call is used to create the shared memory segment and generate the associated system data structure or to gain access to an existing segment.**
- **The shared memory segment and the system data structure are identified by a unique shared memory identifier that the shmget system call returns.**

Include File(s)	<sys/ipc.h> <sys/shm.h>	Manual Section	2
Summary	int shmget(key_t key, int size,int shmflg);		
Return	Success	Failure	Sets errno
	Shared memory identifier.	-1	Yes

Creating a Shared Memory Segment

- **The shmget system call creates a new shared memory segment if**
  - The value for its 1st argument, key, is the symbolic constant IPC_PRIVATE, or
  - the value key is not associated with an existing shared memory identifier and the IPC_CREAT flag is set as part of the shmflg argument or
  - the value key is not associated with an existing shared memory identifier and the IPC_CREAT along with the IPC_EXCL flag have been set as part of the shmflg argument.
- **ftok() can be used to generate a key value.**
- **The argument size determines the size in bytes of the shared memory segment.**
- **If we are using shmget to access an existing shared memory segment, size can be set to 0, as the segment size is set by the creating process.**

Creating a Shared Memory Segment

- **The last argument for shmget, shmflg, is used to indicate segment creation conditions (e.g., IPC_CREAT, IPC_EXCL) and access permissions (stored in the low order 9 bits of shmflg).**
- **The shmget system call does not entitle the creating process to actually use the allocated memory.**
- **It merely reserves the requested memory.**
- **To be used by the process, the allocated memory must be attached to the process using a separate system call.**

3-2 Communication

Task Communication

- **Goal**
  - Pass information to other tasks
  - Coordinate activities in multithreaded embedded application
- **Communication can be signal-centric, data-centric, or both**
  - *signal-centric communication*:
    - all necessary information is conveyed within the event signal itself
  - *data-centric communication*: information is carried within the transferred data
  - Combination: data transfer accompanies event notification
- **Loosely coupled communication: if communication involves data flow and is unidirectional**
  - Data producer does not require a response from the consumer
  - Example: ISR for an I/O device retrieves data from a device and routes data to a dedicated task

Tightly-Coupled Communication

- ***Tightly coupled comm*: data movement is bidirectional**
  - Data producer synchronously waits for a response to its data transfer before resuming execution
  - Or, response is returned asynchronously while data producer continues its function

  - Task #1 sends data to task #2 using message queue #2 and waits for confirmation to arrive at message queue #1
    - Data communication is bidirectional
  - Why use message queue?
    - Task #1 can send multiple messages to task #2
      - Task #1 can continue sending messages while waiting for confirmation to arrive on message queue #2

Why Communication?

- **Transferring data from one task to another**
  - Goal
    - For one task to transfer data to another task
    - Data dependency might exists between tasks: one task is data producer and another task is data consumer
  - Example
    - Task A waits for data to arrive from queues or pipes or shared mem
      - Data producer can be either an ISR or another task
      - Data consumer is Task A
- **Signaling the occurrences of events between tasks**
  - Goal
    - For one task to signal the occurrences of events to another task
    - Either physical devices or other tasks can generate events
      - Task/ISR that is responsible for events can signal occurrences of these events to other tasks (Data might accompany event signals)
  - Example
    - Timer chip ISR notifies another task of the passing of a time tick

Why Communication? (Cont.)

- **Allowing one task to control the execution of other tasks**
  - Goal
    - For one task to control the execution of other tasks
    - Process Control: tasks can have a master/slave relationship
  - Example
    - Master task controls individual subordinate tasks
    - Each subtask is responsible for a component, such as various sensors of the control system
    - Master task sends commands to the subordinate tasks to enable or disable sensors
    - Data flow can be either unidirectional or bidirectional
- **Synchronizing activities**
  - Goal
    - To synchronize activities
  - Example
    - Activity Synchronization
      - Multiple tasks are waiting at barrier: each waits for a signal from last task to continue its own execution
      - It is insufficient to notify tasks that final computation has completed; additional info (e.g. computation results) must also be conveyed

Why Communication? (Cont.)

- **Implementing custom sync protocols for resource sharing**
  - Goal
    - To implement additional sync protocols for resource sharing
  - Multithreaded tasks can implement custom, more-complex resource sync protocols on top of the system-supplied sync primitives

3-3 Synchronization

Synchronization

- **Synchronization is classified into two categories**
  - *resource synchronization*
    - Resource synchronization determines whether access to a shared resource is safe, and, if not, when it will be safe
  - *activity synchronization*
    - Activity synchronization determines whether the execution of a multithreaded program has reached a certain state and, if it hasn't, how to wait for and be notified when this state is reached

Resource Synchronization

- **Resource Sync: maintain integrity of shared resources for access by multiple tasks**
  - Closely associated with critical sections and mutual exclusions.
- ***Mutual exclusion (Mutex)*: only one task at a time can access a shared resource**
- ***Critical section (CS)*: section of code from which the shared resource is accessed**
- **Example**
  - Two tasks trying to access shared memory

- One task (the sensor task) periodically receives data from a sensor and writes the data to shared memory
- Second task (the display task) periodically reads from shared memory and sends the data to a display

Resource Synchronization (Cont.)

- **Problem arise if access is not exclusive: multiple tasks can access the resource simultaneously**
  - If sensor task has not completed writing data to shared memory before display task tries to display data
- **Critical Section (CS)**
  - Code in sensor task that writes input data to memory is CS
  - Code in display task that reads data from memory is CS
    - These two CSs are called *competing critical sections*: access the same shared resource
    - Mutual exclusion algorithm ensures that execution of CS is not interrupted by competing CSs of other tasks
- **Synchronize access: use a client-server model**
  - *Resource server* is responsible for synchronization
  - Access requests are made to resource server
  - Resource server must grant permission to requestor before requestor can access shared resource
    - Based on pre-assigned rules or run-time heuristics

Activity Synchronization

- ***Activity synchronization* is also called *condition synchronization* or *sequence control***
  - Ensures that correct execution order among cooperating tasks is used
    - Can be either synchronous or asynchronous
  - Barrier Sync (柵欄), Rendezvous Sync (約會)

Barrier Synchronization

- ***Barrier synchronization* compromises**
  - A task posts its arrival at the barrier
  - Task waits for other participating tasks to reach the barrier
  - Task receives notification to proceed beyond the barrier
- **For Example,**
  - Divide a complex computation and distribute parts among multiple tasks
    - Some parts of this complex computation are I/O bound
    - Other parts are CPU intensive
    - Others are mainly floating-point operations
  - Partial results must be collected from various tasks for final calculation
  - Barrier: points at collection and duration of final calculation
  - One task can finish its partial computation b/f other tasks complete theirs
  - This task must wait for all other tasks to complete computations b/f it can continue

Barrier Synchronization (Cont.)

- **Assume**
  - Five tasks participates in barrier synchronization
  - Tasks complete partial execution and reach the barrier at various times
    - Each task must wait at barrier until all other tasks have reached it
  - Last task (T5) to reach barrier broadcasts a notification to other tasks
    - All tasks cross the barrier at the same time (conceptually)

Rendezvous Synchronization

- ***Rendezvous synchronization*** **is an execution point where two tasks meet (entry)**
  - Main difference between the barrier and the rendezvous is
    - Barrier allows activity synchronization among two or more tasks
    - Rendezvous synchronization is between two tasks
  - *Entry* is constructed as a function call
    - One task defines its entry and makes it public
    - Any task with knowledge of this entry can call it as an ordinary function call
    - Task that defines the entry accepts the call, executes it, and returns results to caller
    - Caller establishes a rendezvous with the task that defined the entry

Rendezvous Sync (Cont.)

- **Rendezvous sync is similar to sync using event-registers**
  - Caller is blocked if that call is not yet accepted
  - Task that accepts an entry call is blocked when no other task has issued the entry call
- **Rendezvous differs from event-register in that bidirectional data movement (input parameters/output results) is possible**
- **A derivative of rendezvous sync: *simple rendezvous***
  - Uses kernel primitives to achieve sync.
    - Semaphores or message queues, instead of the entry call
  - Two tasks can implement a simple rendezvous without data passing by using two binary semaphores

```
Both semaphores are initialized to 0
When task #1 reaches rendezvous,
   it gives sem#2, then gets on sem#1
When task #2 reaches rendezvous,
   it gives sem#1, then gets on sem#2
Task #1 has to wait on sem#1 b/f task #2
arrives, and vice versa
```

Implementing Barriers

- **How to implement a barrier-synchronization?**
  - Let's use a mutex and a condition variable for implementation

```
typedef struct {
   mutex_t br_lock; /* guarding mutex */
   cond_t br_cond; /* condition variable */
   int br_count; /* num of tasks at the barrier */
   int br_n_threads; /* num of tasks participating in barrier sync */
} barrier_t;
```

Implementing Barriers (Cont.)

- **Each task invokes the function barrier for barrier sync**
  - Line #2: acquire mutex for br_count and br_n_threads
  - Line #3: update #(waiting tasks at the barrier)
  - Line #4: check if all of the participating tasks have reached the barrier
  - Line #5: caller waits on the condition variable
  - Line #6: if caller is the last task, it resets the barrier
  - Line #7: last task notifies all other tasks that barrier sync is complete

Resource Synchronization Methods

- **Semaphores**
- **Mutexes**
- **Interrupt locking**
- **Preemption locking**

Interrupt Locks

- ***Interrupt locking* (disabling system interrupts) is used to sync exclusive access to shared resources b/w tasks & ISRs**
  - Some CPU architectures allow for interrupt-level lock
    - Interrupt lock level is specified so that async events at or below the level of disabled interrupt are blocked for the duration of the lock
  - Other CPU architecture allow only coarse-grained locking
    - All system interrupts are disabled
- **When interrupts are disabled at certain levels**
  - It guarantees that current task continues to execute until it voluntarily relinquishes control
    - Interrupt locking can also be used to sync access to shared resources between tasks
- **Side effects of frequently use of interrupt locks**
  - Missed external events (resulting in data overflow)
  - Clock drift (resulting in missed deadlines)
  - Introduce indeterminism into system
    - Duration of interrupt locks should be short
    - Interrupt locks should be used only when necessary to guard a task-level critical region from interrupt activities

Interrupt Locks (Cont.)

- **A task that enabled interrupt locking must avoid blocking**
  - Behavior of a task making a blocking call is implementation dependent
    - Some RTOSes block the calling task and then re-enable the system interrupts
      - Kernel disables interrupts again on behalf of the task after the task is ready to be unblocked
    - System can hang forever in RTOSes that do not support this feature

Preemption Locks

- ***Preemption locking*** **(disabling the kernel scheduler) is used in resource synchronization**
  - Many RTOS kernels support priority-based, preemptive task scheduling
  - Task disables the kernel preemption when it enters its critical section and re-enables the preemption when finished
  - Executing task cannot be preempted while preemption lock is in effect
- **Preemption locking is more acceptable than interrupt locking**
  - Preemption locking introduces the possibility for priority inversion
    - Even interrupts are enabled while preemption locking is in effect
      - Actual servicing of event is usually delayed to a dedicated task
      - ISR must notify the dedicated task that such an event has occurred
- **Dedicated task usually executes at a high priority**
  - However, this Higher priority task cannot run while another task is inside a critical region that a preemption lock is guarding
    - Problem: higher priority tasks cannot execute, even when they are totally unrelated to CS that preemption lock is guarding
      - Indeterminism is similar to that caused by the interrupt lock

Preemption Locks (Cont.)

- **Example**
  - Consider two medium-priority tasks that share a CS and that use preemption locking as the sync primitive and an unrelated print daemon runs at a much higher priority
  - Printer daemon cannot send a command to the printer to eject one page and feed the next while either of the medium tasks is inside the critical section
    - Result: garbled output or output mixed from multiple print jobs
- **Benefit of preemption locking**
  - It allows accumulation of asynchronous events instead of deleting them
  - I/O device is maintained in a consistent state because its ISR can execute
- **When a task makes a blocking call while preemption is disabled, and another task is scheduled to run**
  - Scheduler disables preemption after original task is ready to resume execution

Critical Section Revisited

- **CS of a task is a section of code that accesses a shared resource**
- **Competing CS is a section of code in another task that accesses the same resource**
  - If these tasks do not have real-time deadlines and guarding the critical section is used only to ensure exclusive access
    - Duration of CS is not important
- **Imagine that a system has two tasks**
  - Task A performs some calculations and stores the results in a shared variable
  - Task B reads the shared variable and displays its value
  - Using mutual exclusion algorithm to guard the CS ensures that each task has exclusive access to the shared variable
    - No real-time requirements, and the only constraint placed on the two tasks is: write() precedes read() on the shared variable
- **If Task C without a competing CS exists in system but does have real-time deadlines to meet**
  - Task C must be allowed to interrupt either of the other two tasks, regardless of whether the task is in its critical section,
    - To guarantee overall system correctness
  - Duration of CS of the first two tasks can be long, and higher priority task should be allowed to interrupt

Critical Section Revisited (Cont.)

- **If Task A, B have real-time deadlines and the time needed to complete their CSs impacts whether the tasks meet their deadlines, this CS should run to completion without interruption**
  - Preemption lock becomes useful in this situation.
- **It is important to evaluate the criticality of the CS and the overall system impact before deciding on which mutual exclusion algorithm to use for guarding a CS**
- **Solution to the mutual exclusion problem should satisfy the following conditions:**
  - Only one task can enter its critical section at any given time
  - Fair access to shared resource by multiple competing tasks is provided
  - One task executing its CS must not prevent another task executing a non-competing CS

Common Practical Design Patterns

- **Synchronous Activity Synchronization**
  - Task-to-task synchronization using binary semaphores
  - ISR-to-task synchronization using binary semaphores
  - Task-to-task synchronization using event registers
  - ISR-to-task synchronization using event registers
  - ISR-to-task synchronization using counting semaphores
  - Simple rendezvous with data passing
- **Asynchronous Event Notification Using Signals**
- **Resource Synchronization**
  - Shared Memory with Mutexes
  - Shared Memory with Interrupt Locks
  - Shared Memory with Preemption Locks
  - Sharing Multiple Instances of Resources Using Counting Semaphores and Mutexes

Synchronous Activity Synchronization

- **Task-to-task synchronization using binary semaphores**
- **ISR-to-task synchronization using binary semaphores**
- **ISR-to-task synchronization using counting semaphores**
- **Simple rendezvous with data passing**

Task-to-Task Synchronization Using Binary Semaphores

- **Two tasks synchronize their activities using a binary semaphore**
  - Initial value of the binary semaphore is 0
  - Task #2 has to wait for task #1 to reach an execution point
    - Task #1 signals to task #2 its arrival at the execution point by giving the semaphore and changing the value to 1
  - At this point, task #2 can run if it has higher priority
  - Value of the binary semaphore is reset to 0 after the synchronization
  - Task #2 has execution dependency on task #1

ISR-to-Task Synchronization Using Binary Semaphores

- **Task & ISR sync their activities using a binary semaphore**
  - Initial value of the binary semaphore is 0
  - Task has to wait for ISR to signal occurrence of an asynchronous event
    - When event occurs and the associated ISR runs, it signals to the task by giving the semaphore and changing the value to 1
  - ISR runs to completion before the task gets to resume execution
  - Value of semaphore is reset to 0 after the task resumes execution

ISR-to-Task Synchronization Using Counting Semaphores

- **A counting semaphore is used to accumulate event occurrences and for task signaling**
  - Value of the counting semaphore increments by one each time the ISR gives the semaphore
  - Value is decremented by one each time the task gets the semaphore
  - Task runs as long as the counting semaphore is non-zero

Simple Rendezvous with Data Passing

- **Two tasks can exchange data at the rendezvous point using two message queues**
  - Each message queue can hold a maximum of one message
  - Both message queues are initially empty
  - When task #1 reaches the rendezvous, it puts data into message queue #2 and waits for a message to arrive on message queue #1
  - When task #2 reaches the rendezvous, it puts data into message queue #1 and waits for data to arrive on message queue #2
  - Task #1 has to wait on message queue #1 before task #2 arrives, and vice versa, thus achieving rendezvous synchronization with data passing

Resource Synchronization

- **Shared Memory with Mutexes**
- **Shared Memory with Interrupt Locks**
- **Shared Memory with Preemption Locks**
- **Sharing Multiple Instances of Resources Using Counting Semaphores and Mutexes**

Shared Memory with Mutexes

- **Task #1 and task #2 access shared memory using a mutex for synchronization**
- **Each task must first acquire the mutex before accessing the shared memory**
  - Task blocks if the mutex is already locked, indicating that another task is accessing the shared memory
  - Task releases the mutex after it completes its operation on the shared memory

Shared Memory with Interrupt Locks

- **ISR transfers data to the task using shared memory**
  - ISR puts data into the shared memory, and the task removes data from the shared memory and subsequently processes it
  - Interrupt lock is used for synchronizing access to the shared memory
  - Task must acquire and release the interrupt lock to avoid the interrupt disrupting its execution

  - ISR does not need to be aware of the existence of the interrupt lock
    - Unless nested interrupts are supported (i.e., interrupts are enabled while an ISR executes) and multiple ISRs can access the data

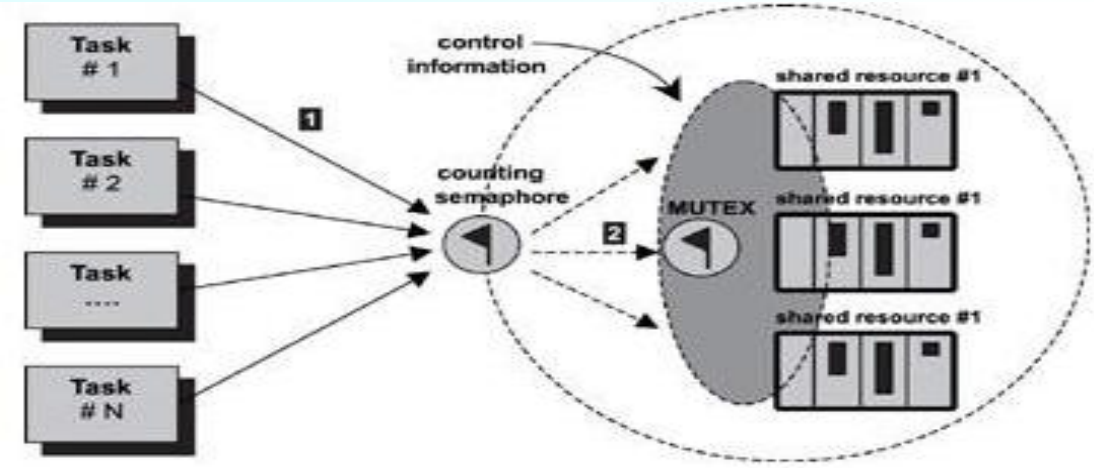
Shared Memory with Preemption Locks

- **Two tasks transfer data to each other using shared memory**
  - Each task is responsible for disabling preemption before accessing the shared memory
  - Unlike using a binary semaphore or a mutex lock, no waiting is involved when using a preemption lock for synchronization

Sharing Multiple Instances of Resources Using Counting Semaphores and Mutexes

- **Scenario**
  - *N* tasks share *M* instances of a single resource type (*M* printers)
    - Counting semaphore tracks the number of available resource instances at any given time
    - Counting semaphore is initialized with the value *M*
    - Each task must acquire the counting semaphore before accessing the shared resource
    - By acquiring the counting semaphore, the task effectively reserves an instance of the resource
      - Having the counting semaphore alone is insufficient. Typically, a control structure associated with the resource instances is used
    - Control structure maintains information such as which resource instances are in use and which are available for allocation
    - Control information is updated each time a resource instance is either allocated to or released by a task
      - A mutex is deployed to guarantee that each task has exclusive access to the control structure
    - After a task successfully acquires the counting semaphore, the task must acquire the mutex before the task can either allocate or free an instance

N Task vs. *M* Same Shared Resources



4
I/O Subsystem

I/O…

- **All embedded systems include some form of I/O operations**
  - I/O operations are performed over different types of I/O devices
    - Vehicle dashboard display, touch screen on a PDA
    - Hard disk of a file server
    - NIC
- **Embedded system is designed to handle special requirements associated with a device**
  - Cell phone, pager, and a handheld MP3 player
- **For software developer, I/O operations imply**
  -
  -
  -
  -
- **For RTOS, I/O operations imply**
  -
  -
  -

Basic I/O Concepts

- **Combination of I/O devices, device drivers, and I/O subsystem comprises overall I/O system in embedded env.**
- **Purpose of I/O subsystem**
  - Hide device-specific info from kernel and application developer
  - Provide a uniform access method to peripheral I/O devices of the system
- **I/O vs. System**
  - Each descending layer adds additional detailed information to the architecture needed to manage a given device

How to Access I/O Devices?

- **I/O Device Hardware**
  - Range from low-bit rate serial lines to hard drives and gigabit network interface adaptors
- **Device Control Registers**
  - Located on the CPU board or in the devices themselves
  - I/O devices must be initialized through device control registers (external to CPU)
  - Device registers are accessed & programmed for data transfer requests: device control
  - Developer must determine if the device is port mapped or memory mapped
    - This determines the way to access an I/O device
- **If I/O device address space is separate from sys memory address space**
  - Special processor instructions (IN & OUT) are used to transfer data between the I/O device and a microprocessor register or memory.

Port-Mapped I/O

- **I/O device address is referred to as the port number**
  - This form of I/O is called port-mapped I/O or isolated I/O
    - Because memory space is isolated from I/O space
    - Entire memory address space is available for application use
- **Devices are programmed to occupy a range in I/O address space**
  - Each device is on a different I/O port
  - I/O ports are accessed through special processor instructions
    - Actual physical access is accomplished through special hardware circuitry

Memory-Mapped I/O

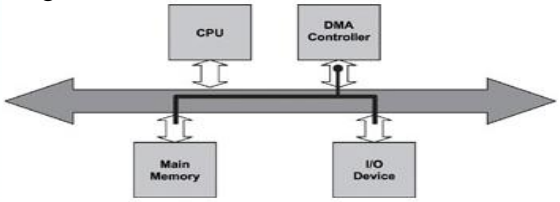
- **Memory-mapped I/O: device address is part of system memory address space**
  - I/O device is treated as if it were another memory location
- **Any machine instruction may be used to access I/O devices**
  - Instructions to transfer data between memory location and processor
  - Instructions to transfer data between two memory locations
- **Memory address space occupied by I/O cannot be used for application**
  - Memory-mapped I/O space does not necessarily begin at offset 0
  - Can be mapped anywhere inside the address space
  - This is implementation-dependent

Where to Map?

- **Tables describing mapping of a device's internal registers are available in device hardware data book**
  - Device registers appear at different offsets in this map
  - Sometimes information is presented in the 'base + offset' format
    - Addresses in the map are relative
      - Port-mapped I/O: offset must be added to start address of I/O space
      - Memory-mapped I/O: offset must be added to base address of system memory space for memory-mapped I/O

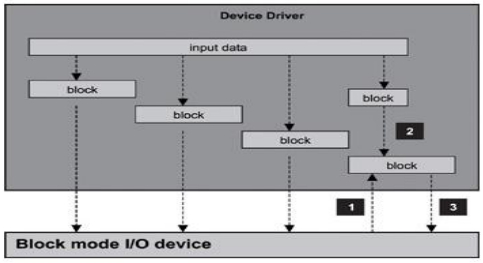
DMA I/O

- **Processor involves to the work in port-/memory-mapped I/O**
- **Data transfer between device and system involves**
  - Transferring data between device and processor register and
  - Then from the processor register to memory
  - Transfer speed might not meet needs of high-speed I/O devices because of additional data copy
- **Direct memory access (DMA) chips/controllers allow device to access memory directly without involving processor**
  - Processor sets up DMA controller before a data transfer
    - Specify source/destination address, and data length to DMA controller
  - Processor is bypassed during data transfer
  - Transfer speed depends on
    - Transfer speed of I/O device
    - Speed of the memory device
    - Speed of the DMA controller



Character-Mode vs. Block-Mode

- **I/O devices are classified as character-/block-mode devices**
  - How the device handles data transfer with the system
- **Character-mode devices: for unstructured data transfers**
  - Data transfers take place in serial, one byte at a time
  - Character-mode devices are simple devices: serial interface or keypad
    - Driver buffers data in cases if transfer rate from system to device is faster than device's handling rate
- **Block-mode devices: transfer one block at time (64B per transfer)**
  - Underlying h/w imposes block size
  - Some structure must be imposed on data
  - Some transfer protocol must be enforced
  - Sometimes, block-mode device driver need to perform additional work for each read/write operation



Block-Mode Devices

- **On servicing a write operation with large amounts of data,**
  - Method I
    - Driver first divide data into many blocks w/ device-specific block size
      - Last block often is smaller than the normal device block size
    - Each block is transferred to the device in separate write requests
      - First three are straightforward write operations
      - Since last block has different size, driver must handle it differently
        - » Method to process last block is device specific (e.g. padding)
  - Method II
    - Accumulate data in driver cache
    - Perform write after enough data has accumulated for required size
      - Minimizes number of device accesses
      - Disadvantages:
        - » Driver is more complex: driver must know if cached data satisfy read operation
        - » Delayed write can also cause data loss if a failure occurs and if driver is shut down and unloaded ungracefully
        - » Data caching implies data copying: lower I/O performance

I/O Subsystem

- **Driver can provide a driver-specific I/O APIs to application**
  - Application need to know nature/restriction of I/O device
    - API set is driver-/implementation-dependent: difficult to port
      - Solution: embedded systems often include *I/O subsystem*
- **I/O subsystem**
  - Defines a standard set of functions (API) for I/O operations
    - Hide device peculiarities from applications
    - Provide uniform I/O to applications
      - All I/O device drivers conform to and support this function set
- **Concepts**
  - Driver implements each function in the standard set of I/O API
  - Driver exports the set of functions to the I/O subsystem
  - Driver sets up an association between I/O subsystem API set and corresponding device-specific I/O calls
  - Driver loads device and makes the association known to I/O subsystem

Mapping Generic Functions to Driver Functions

- **Individual device drivers provide actual implementation of each function in uniform I/O API set**
- **I/O function mapping**
  - I/O subsystem-defined API set needs to be mapped into a driver-specific function set for any driver supports uniform I/O
  - Functions that begin with *driver_* prefix refer to implementations that are specific to a device driver

```
typedef struct
{
  int (*Create)( );
  int (*Open) ( );
  int (*Read)( );
  int (*Write) ( );
  int (*Close) ( );
  int (*Ioctl) ( );
  int (*Destroy) ( );
} UNIFORM_IO_DRV;
```

Mapping Process

- **Mapping process involves initializing each function pointer with the address of an associated internal driver function**
  - Internal driver functions can have any name if they are correctly mapped

```
UNIFORM_IO_DRV ttyIOdrv;
ttyIOdrv.Create = tty_Create;
ttyIOdrv.Open = tty_Open;
ttyIOdrv.Read = tty_Read;
ttyIOdrv.Write = tty_Write;
ttyIOdrv.Close = tty_Close;
ttyIOdrv.Ioctl = tty_Ioctl;
ttyIOdrv.Destroy = tty_Destroy;
```

Uniform I/O Driver Table

- **I/O subsystem usually maintains a *uniform I/O driver table***
  - Driver can be installed into or removed from this driver table by using the utility functions that the I/O subsystem provides
  - Each row is a unique I/O driver that supports the defined API set
  - First column is a generic name used to associate uniform I/O driver with a particular type of device
    - Example: I/O driver is provided for serial line terminal device (tty)
  - These pointers are written to table when driver is installed in subsystem
  - When utility function is called, reference to newly created driver table entry is returned to caller

Associating Devices with Device Drivers

- **I/O subsystem tracks virtual instances using *device table***
  - Newly created virtual instance is given a unique name and is inserted into the device table
  - Each entry in the device table holds generic and instance-specific info
    - Generic: unique name of device instance & a reference to driver
      - tty0 (device name): I/O device is a serial terminal device and is the first instance created in system
    - instance-specific: mem block allocated by driver to hold instance-specific data
      - » Driver is the only entity that accesses/interprets the data
  - Reference to newly created device entry is returned to caller of create function
  - Subsequent calls to open and destroy functions use this reference.

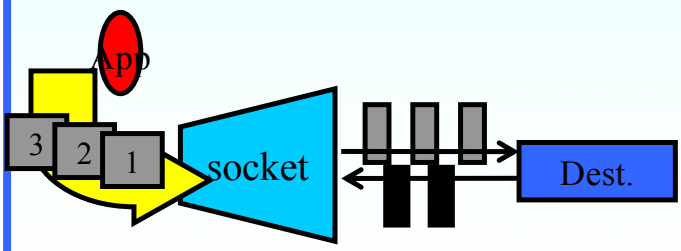
5
Inter-Machine Communication: Socket

Socket Programming

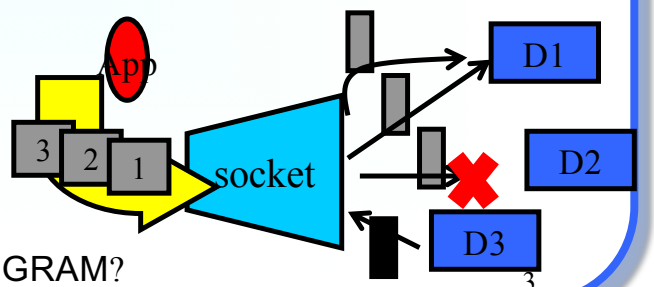
- **What is a socket?**
  - An interface between application and network
    - The application creates a socket
    - The socket *type* dictates the style of communication
      - reliable vs. best effort
      - connection-oriented vs. connectionless
  - Once configured the application can
    - pass data to the socket for network transmission
    - receive data from the socket (transmitted through the network by some other host)

Two essential types of sockets

- **SOCK_STREAM**
  - a.k.a. TCP
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional



- **SOCK_DGRAM**
  - a.k.a. UDP
  - unreliable delivery
  - no order guarantees
  - no notion of "connection" – app indicates dest. for each packet
  - can send or receive



Q: why have type SOCK_DGRAM?

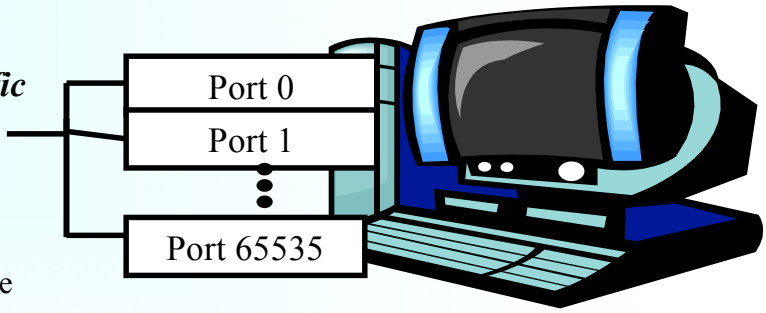
Socket Creation in C: socket

- **int s = socket(domain, type, protocol);**
  - s: socket descriptor, an integer (like a file-handle)
  - domain: integer, communication domain
    - e.g., PF_INET (IPv4 protocol) – typically used
  - type: communication type
    - SOCK_STREAM: reliable, 2-way, connection-based service
    - SOCK_DGRAM: unreliable, connectionless,
    - other values: need root permission, rarely used, or obsolete
  - protocol: specifies protocol (see file /etc/protocols for a list of options) - usually set to 0

- **NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!**

Ports

- **Each host has 65,536 ports**
- **Some ports are *reserved for specific apps***
  - 20,21: FTP
  - 23: Telnet
  - 80: HTTP
  - see RFC 1700 (about 2000 ports are reserved)



- A socket provides an interface to send data to/from the network through a port

Addresses, Ports and Sockets

- **Like apartments and mailboxes**
  - You are the application
  - Your apartment building address is the address
  - Your mailbox is the port
  - The post-office is the network
  - The socket is the key that gives you access to the right mailbox (one difference: assume outgoing mail is placed by you in your mailbox)

- **Q: How do you choose which port a socket connects to?**

The bind function

- **associates and (can exclusively) reserves a port for use by the socket**
- **int status = bind(sockid, &addrport, size);**
  - status: error status, = -1 if bind failed
  - sockid: integer, socket descriptor
  - addrport: struct sockaddr, the (IP) address and port of the machine (address usually set to INADDR_ANY – chooses a local address)
  - size: the size (in bytes) of the addrport structure
- **bind can be skipped for both types of sockets. When and why?**

Skipping the bind

- **SOCK_DGRAM:**
  - if only sending, no need to bind. The OS finds a port each time the socket sends a pkt
  - if receiving, need to bind
- **SOCK_STREAM:**
  - destination determined during conn. setup
  - don't need to know port sending from (during connection setup, receiving end is informed of port)

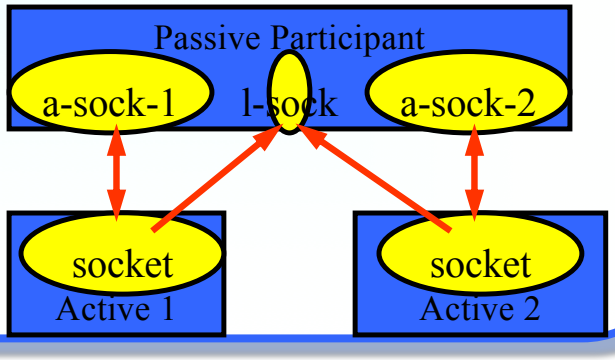
Connection Setup (SOCK_STREAM)

- **Recall: no connection setup for SOCK_DGRAM**
- **A connection occurs between two kinds of participants**
  - passive: waits for an active participant to request connection
  - active: initiates connection request to passive side
- **Once connection is established, passive and active participants are "similar"**
  - both can send & receive data
  - either can terminate the connection

Connection setup cont'd

- **Passive participant**
  - step 1: listen (for incoming requests)
  - step 3: accept (a request)
  - step 4: data transfer
- **The accepted connection is on a new socket**
- **The old socket continues to listen for other active participants**
- **Why?**

- **Active participant**
  - step 2: request & establish connection
  - step 4: data transfer



Connection setup: listen & accept

- **Called by passive participant**
- **int status = listen(sock, queuelen);**
  - status: 0 if listening, -1 if error
  - sock: integer, socket descriptor
  - queuelen: integer, # of active participants that can "wait" for a connection
  - listen is **non-blocking**: returns immediately
- **int s = accept(sock, &name, &namelen);**
  - s: integer, the new socket (used for data-transfer)
  - sock: integer, the orig. socket (being listened on)
  - name: struct sockaddr, address of the active participant
  - namelen: sizeof(name): value/result parameter
    - must be set appropriately before call
    - adjusted by OS upon return
  - accept is **blocking**: waits for connection before returning

connect call

- **int status = connect(sock, &name, namelen);**
  - status: 0 if successful connect, -1 otherwise
  - sock: integer, socket to be used in connection
  - name: struct sockaddr: address of passive participant
  - namelen: integer, sizeof(name)
- **connect is blocking**

Sending / Receiving Data

- **With a connection (SOCK_STREAM):**
  - int count = send(sock, &buf, len, flags);
    - count: # bytes transmitted (-1 if error)
    - buf: char[], buffer to be transmitted
    - len: integer, length of buffer (in bytes) to transmit
    - flags: integer, special options, usually just 0
  - int count = recv(sock, &buf, len, flags);
    - count: # bytes received (-1 if error)
    - buf: void[], stores received bytes
    - len: # bytes received
    - flags: integer, special options, usually just 0
  - Calls are **blocking** [returns only after data is sent (to socket buf) / received]

Sending / Receiving Data (cont'd)

- **Without a connection (SOCK_DGRAM):**
  - int count = sendto(sock, &buf, len, flags, &addr, addrlen);
    - count, sock, buf, len, flags: same as send
    - addr: struct sockaddr, address of the destination
    - addrlen: sizeof(addr)
  - int count = recvfrom(sock, &buf, len, flags, &addr, &addrlen);
    - count, sock, buf, len, flags: same as recv
    - name: struct sockaddr, address of the source
    - namelen: sizeof(name): value/result parameter
- **Calls are blocking [returns only after data is sent (to socket buf) / received]**

close

- **When finished using a socket, the socket should be closed:**
- **status = close(s);**
  - status: 0 if successful, -1 if error
  - s: the file descriptor (socket being closed)
- **Closing a socket**
  - closes a connection (for SOCK_STREAM)
  - frees up the port used by the socket

The struct sockaddr

- **The generic:**

```
struct sockaddr {
        u_short sa_family;
        char sa_data[14];
};
```

  - sa_family
    - specifies which address family is being used
    - determines how the remaining 14 bytes are used

- **The Internet-specific:**

```
struct sockaddr_in {
        short sin_family;
        u_short sin_port;
        struct in_addr sin_addr;
        char sin_zero[8];
};
```

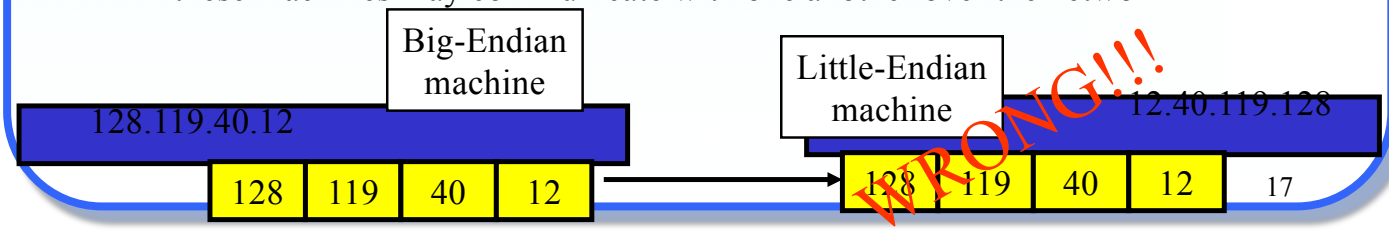
  - sin_family = AF_INET
  - sin_port: port # (0-65535)
  - sin_addr: IP-address
  - sin_zero: unused

Address and port byte-ordering

- **Address and port are stored as integers**
  - u_short sin_port; (16 bit)
  - in_addr sin_addr; (32 bit)

```
struct in_addr {
        u_long s_addr;
};
```

- Problem:
  - different machines / OS's use different word orderings
    - little-endian: lower bytes first
    - big-endian: higher bytes first
  - these machines may communicate with one another over the network



Solution: Network Byte-Ordering

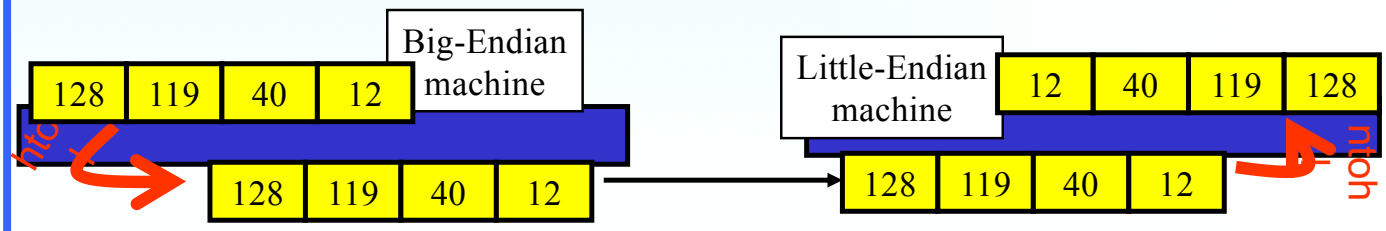
- **Definition:**
  - Host Byte-Ordering: the byte ordering used by a host (big or little)
  - Network Byte-Ordering: the byte ordering used by the network – always **BIG**-endian
- **Any words sent through the network should be converted to Network Byte-Order prior to transmission (and back to Host Byte-Order once received)**
- **Q: should the socket perform the conversion automatically?**

- Q: Given big-endian machines don't need conversion routines and little-endian machines do, how do we avoid writing two versions of code?

UNIX's byte-ordering funcs

- u_long htonl(u_long x);
- u_short htons(u_short x);
- u_long ntohl(u_long x);
- u_short ntohs(u_short x);

- On big-endian machines, these routines do nothing
- On little-endian machines, they reverse the byte order



- Same code would have worked regardless of endian-ness of the two machines

Dealing with blocking calls

- **Many of the functions we saw block until a certain event**
  - accept: until a connection comes in
  - connect: until the connection is established
  - recv, recvfrom: until a packet (of data) is received
  - send, sendto: until data is pushed into socket's buffer
    - Q: why not until received?
- **For simple programs, blocking is convenient**
- **What about more complex programs?**
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing

Dealing w/ blocking (cont'd)

- **Options:**
  - create multi-process or multi-threaded code
  - turn off the blocking feature (e.g., using the fcntl file-descriptor control function)
  - use the select function call.
- **What does select do?**
  - can be permanent blocking, time-limited blocking or non-blocking
  - input: a set of file-descriptors
  - output: info on the file-descriptors' status
  - i.e., can identify sockets that are "ready for use": calls involving that socket will return immediately

select function call

- **int status = select(nfds, &readfds, &writefds, &exceptfds, &timeout);**
  - status: # of ready objects, -1 if error
  - nfds: 1 + largest file descriptor to check
  - readfds: list of descriptors to check if read-ready
  - writefds: list of descriptors to check if write-ready
  - exceptfds: list of descriptors to check if an exception is registered
  - timeout: time after which select returns, even if nothing ready - can be 0 or ∞ (point timeout parameter to NULL for ∞)

To be used with select:

- **Recall select uses a structure, struct fd_set**
  - it is just a bit-vector
  - if bit *i* is set in [readfds, writefds, exceptfds], select will check if file descriptor (i.e. socket) *i* is ready for [reading, writing, exception]
- **Before calling select:**
  - FD_ZERO(&fdvar): clears the structure
  - FD_SET(i, &fdvar): to check file desc. *i*
- **After calling select:**
  - int FD_ISSET(i, &fdvar): boolean returns TRUE iff *i* is "ready"

Other useful functions

- **bzero(char* c, int n): 0's n bytes starting at c**
- **gethostname(char *name, int len): gets the name of the current host**
- **gethostbyaddr(char *addr, int len, int type): converts IP hostname to structure containing long integer**
- **inet_addr(const char *cp): converts dotted-decimal char-string to long integer**
- **inet_ntoa(const struct in_addr in): converts long to dotted-decimal notation**

- **Warning: check function assumptions about byte-ordering (host or network). Often, they assume parameters / return solutions in network byte-order**

Release of ports

- **Sometimes, a "rough" exit from a program (e.g., ctrl-c) does not properly free up a port**
- **Eventually (after a few minutes), the port will be freed**
- **To reduce the likelihood of this problem, include the following code:**

```
#include <signal.h>
void cleanExit(){exit(0);}
```

  - in socket code:

```
signal(SIGTERM, cleanExit);
signal(SIGINT, cleanExit);
```

6
Memory

Dynamic Memory Allocation

- **After program initialization completes, program code, program data, and system stack occupy the physical memory**
- **RTOS or kernel typically uses remaining physical memory for dynamic memory allocation**
  - This area is called the *heap*
- **Memory management facility maintains internal info for a heap in a reserved memory area (*control block*)**
  - Typical internal information includes:
    - Starting address of physical memory block used for dynamic allocation
    - Overall size of this physical memory block
    - Allocation table that indicates which memory areas are in use, which memory areas are free, and the size of each free region

Memory Fragmentation

- **Example I**
  - Heap (256 bytes) is broken into small, fixed-size blocks
  - Each block has a unit size (32 bytes) that is power of two
  - Dynamic memory allocation function, malloc()
    - Input parameter: specifies size of the allocation request in bytes
  - malloc() allocates a larger block: be made up of one or more of the smaller, fixed-size blocks
    - If allocating 100 bytes, returned block has a size of 128 bytes (4 units x 32 bytes/unit)
      - Requestor does not use 28 bytes
        - » Memory fragmentation: internal fragmentation
        - » Internal to allocated block
  - Allocation table represents as a bitmap
    - Each bit means a 32-byte unit

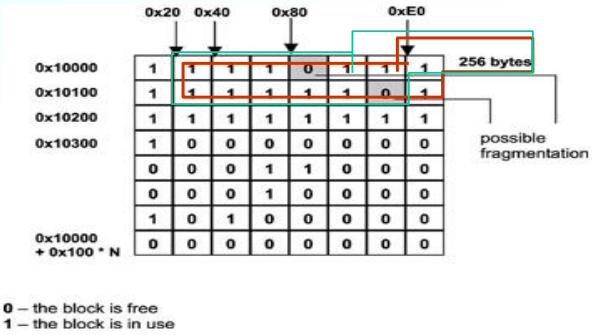
Memory Fragmentation (cont)

- Step 6: two free blocks of 32 bytes
- Step 7: all 3 blocks are combined to form a 128-byte block, instead of maintaining three separate free blocks

Memory Fragmentation (cont)

- **Example II**
  - There are two free 32-byte blocks
    - One block is at address 0x10080, and the other at address 0x101C0
      - Cannot be used for any memory allocation larger than 32 bytes
        - » External fragmentation
        - » Fragmentation exists in table, not within blocks themselves.
- **Solution to External Fragmentation**
  - Compact area adjacent to these 2 blocks
    - From address 0x100A0 to 0x101BF
      - Shifted 32 bytes lower in memory, 0x10080 to 0x1019F

      →Combine 2 free blocks into 1 64-byte block



Memory Compaction

- **Several problems occur with memory compaction**
  - Time-consuming to transfer memory content from one location to another
    - Cost of copy operation depends on length of contiguous blocks
  - Tasks that currently hold ownership of those memory blocks are prevented from accessing those memory locations until transfer completes
    - Compaction is almost never done in practice in embedded designs
    - Free memory blocks are combined only if immediate neighbors
- **Memory compaction is allowed if tasks that own those memory blocks reference blocks using virtual addresses**
  - Compaction is not permitted if tasks hold physical addresses to allocated blocks
- **Memory alignment: architecture-specific constraints**
  - Some architecture requires multi-byte data items (e.g. int or long) to be allocated at addresses that are a power of two
    - Unaligned mem addresses result in bus errors and mem access exceptions

Efficient Memory Manager

- **Memory manager needs to perform the following quickly:**
  - Determine if a free block that is large enough exists to satisfy the allocation request. This work is part of the malloc operation.
  - Update the internal management information. This work is part of both the malloc and free operations.
  - Determine if the just-freed block can be combined with its neighboring free blocks to form a larger piece. This work is part of the free operation.
- **Structure of allocation table is key to efficiency**
  - Structure determines how the operations must be implemented
  - Allocation table is part of the overhead because it occupies memory space that is excluded from application use
- **Minimizing management overhead is another issue in efficiency**

Example Implementation of malloc() and free()

- **Static array of ints (*allocation array*): allocation map**
  - To decide if neighboring free blocks can be merged to form larger block
  - Each entry in this array represents corresponding fixed-size mem block
    - #array entries = #fixed-size mem blocks
      - 1MB mem = 32,768 32-byte blocks
      - →#array entries = 32,768

•Let array index start at 0
•Before allocating: one large free block, consisting of all 12 units
•Contiguous free blocks: positive number is placed in first/last entry representing range
•This num = #free blocks in the range
•Allocated: negative mum in first entry and 0 in last entry
•indicates a range of allocated blocks
•12 is placed in the entries at index 0 and 11
•malloc(3): -3 and 0

Example Implementation of malloc() and free()

- **Allocation array**
  - Indicates which blocks are free
  - Implicitly indicates the starting address of each block:
    - starting address = offset + unit_size*index
  - malloc uses this formula to calculate the starting address of the block
    - If offset = 0x10000, unit size = 0x20 (32), address for index 9
      - 0x10000 + 0x20*9 = 0x10120

Finding Free Blocks Quickly

- **malloc always allocates from the largest available range of free blocks**
  - Allocation array described is not arranged to help malloc perform this task quickly
    - Entries representing free ranges are not sorted by size
  - Finding the largest range always entails an end-to-end search
    - Second data structure is used to speed up the search for the free block that can satisfy the allocation request
      - Size of free blocks within allocation array are maintained using heap data structure
    - Heap data structure is a complete binary tree with one property:
      - Value contained at a node is no smaller than value in any of its child nodes

Heap

- **Size of each free block is the key used for arranging heap**
  - Largest free block is always at the top of the heap
  - malloc() carves allocation out of largest available free block
  - Remaining portion is reinserted into the heap
  - Heap is rearranged as the last step of the memory allocation process
- **Node in heap containing at least two pieces of info**
  - Size of a free range & the starting index in the allocation array
- **malloc() involves the following steps:**
  - Examine heap to see if any free block large enough for allocation request
    - If no such block exists, return an error to the caller
  - Retrieve starting allocation-array index from top of heap
  - Update allocation array by marking the newly allocated block
  - If entire block is used, update heap by deleting largest node
    - Otherwise update the size of the node
  - Rearrange heap array

Heap (cont)

- **Before memory has been allocated, heap has just one node**
  - Entire memory region is available as one, large, free block
- **Heap continues to have a single node either**
  - if memory is allocated consecutively without any free operations or
  - if each free() results in deallocated block merging with its neighbors
- **Heap can be implemented using static array (*heap array*)**
  - Array index begins at 1
  - Six free blocks of 20, 18, 12, 11, 9, and 4 blocks are available
  - Next allocation uses 20-block range regardless of size of malloc()
  - Heap array stores no pointers to child nodes
    - Child-parent relationships are indicated by node positions within array

free()

- **Main operation of free() is to determine if block being freed can be merged w/ its neighbors**
- **Merging rules are**
  - If block's starting index is not 0, check for value of array at (index -1)
    - If value is positive (not negative or 0), this neighbor can be merged
  - If (index + #blocks) does not exceed maximum array index, check for value of array at (index + #blocks)
    - If value is positive, this neighbor can be merged

free() (cont)

- **Example I**
  - Block starting at index 3 is being freed
    - Following rule #1, look at the value at index 2
      - Value is 3 → neighboring block can be merged
      - 3 + 4 = 7
    - Following rule #2, look at the value at index 7
      - Value is -2 → neighboring block cannot be merged
- **Example II**
  - Block at index 7 is being freed
    - Following rule #1, look at the value at index 6
      - Value is 0 → neighboring block is in use
    - Following rule #2, look at the value at index 9
      - Value is -3 → this block is also in use
  - Block at index 3 is being freed…

free() (cont)

- **When a block is freed, heap must be updated accordingly**
- **Free operation involves the following steps:**
  - Update the allocation array and merge neighboring blocks if possible
    - If newly freed block cannot be merged, insert new entry into heap
    - If newly freed block can be merged,
      - Update its heap entry
      - Rearranged heap entry according to its new size
    - If newly freed block can be merged with both neighbors,
      - Delete its heap entry
      - Update other neighboring block
      - Rearranged heap entry according to its new size

Memory Management in Embedded Systems

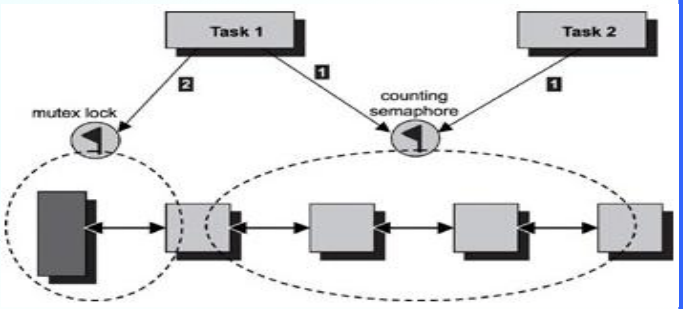
- **Memory space is divided into variously sized memory pools**
  - All blocks of same memory pool have same size
- **Example**
  - Memory space is divided into three pools of block sizes 32, 50, and 128
  - Each memory-pool control structure maintains info
    - block size, total number of blocks, and number of free blocks
  - Memory pools are linked together and sorted by size
    - Finding smallest size adequate for allocation requires searching through this link and examining each control structure for first adequate block size
- **Memory allocation vs. Deallocation**
  - Entry will be removed from pool
  - Entry will be inserted back into pool
  - Pros and Cons (discussion)

Blocking vs. Non-Blocking

- **malloc() & free() do not allow task to block/wait for mem**
  - In many real-time embedded systems, tasks compete for limited memory
  - Sometimes, memory exhaustion condition is only temporary
    - For some tasks when a memory allocation request fails, it must backtrack to an execution checkpoint and restart an operation: expensive operation cost
    - If tasks know that memory congestion condition can occur but only momentarily, the tasks can be designed to be more flexible
    - If such tasks can tolerate the allocation delay, tasks can wait for memory to become available instead of failing or backtracking
- **Example**
  - Ethernet traffic pattern is bursty
  - During traffic burst, tasks in embedded network device can experience temporary memory exhaustion problems
    - Much of the available memory is used for packet reception
  - Sending tasks can wait for condition to subside and resume operations

Blocking vs. Non-Blocking (cont)

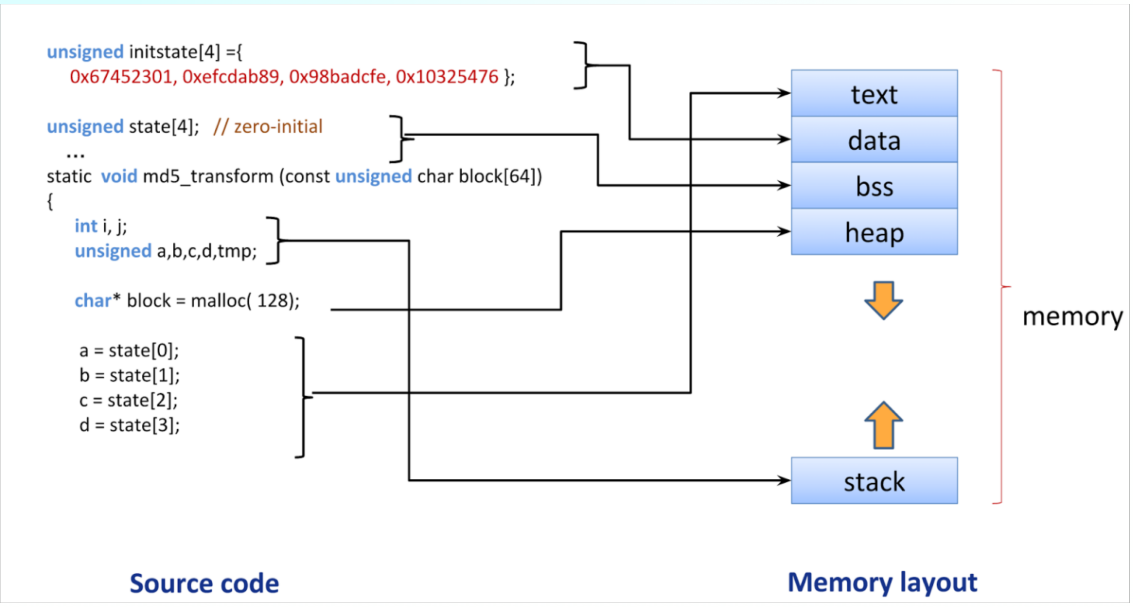
- **Memory allocation function should allow for**
  - Blocking forever
  - Blocking for a timeout period
  - No blocking at all
- **Example**
  - Implement memory allocation function by using counting semaphor and mutex lock
  - These primitives are created for each pool and kept in control structure
  - Counting semaphore is initialized w/ total #blocks on creating mem pool



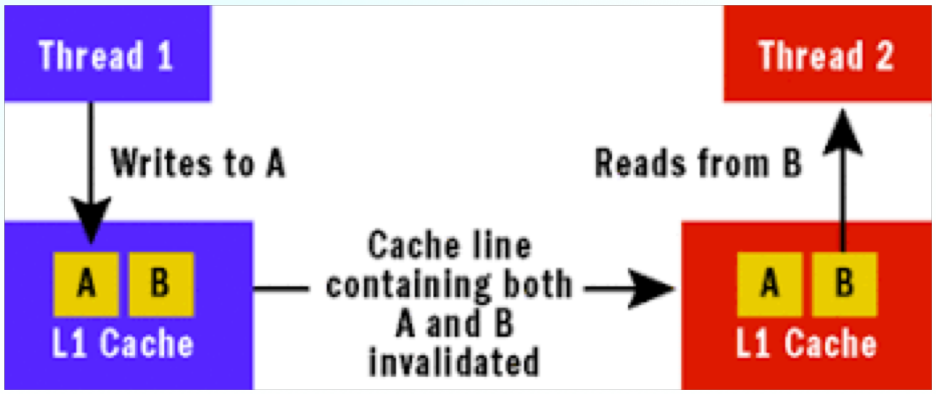
Hardware MMU

- **Virtual memory**
  - Mass storage is made to appear to application
- **Virtual memory address space (*logical address space*) is larger than actual physical memory space**
  - Allows a program larger than physical memory to execute
- ***Memory management unit* (MMU) provides several functions**
  - MMU translates virtual address to physical address for each mem access
  - MMU provides memory protection
- **If MMU is enabled, physical memory is divided into *pages***
- **A set of attributes is associated with each memory page:**
  - whether page contains code (i.e., executable instructions) or data
  - whether page is readable, writable, executable, or a combination of these
  - whether page can be accessed when CPU is not in privileged execution mode, accessed only when CPU is in privileged mode, or both
- **All memory access is done through MMU**

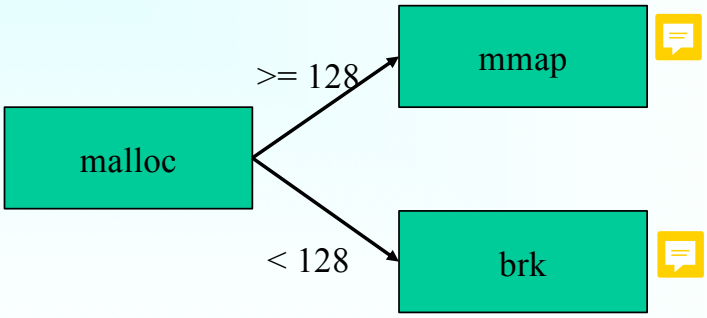
Program Memory Layout



Memory vs. Multithreads

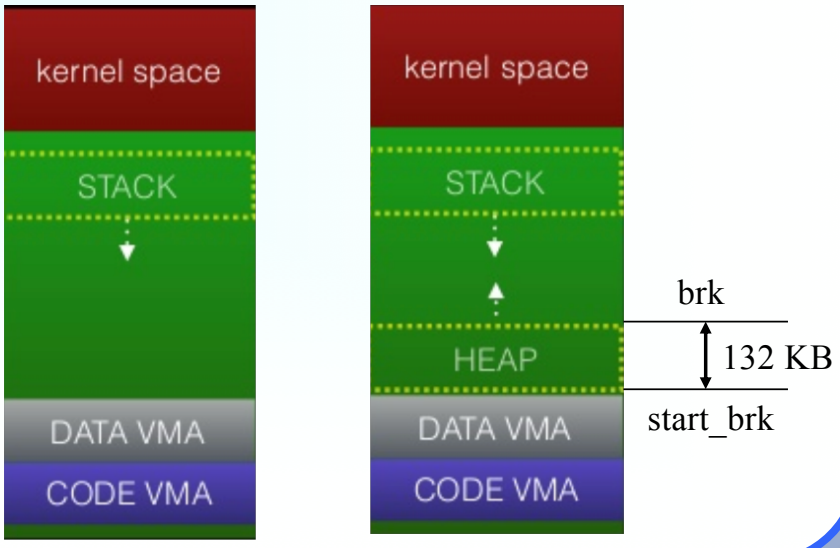


Heap Exploitation



Workflow of malloc()

無論一開始 malloc 多少空間（<128KB），Kernel 都會給 132 KB，這一段叫 main arena

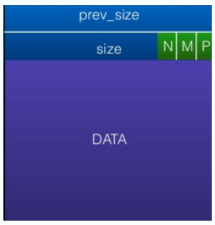


Workflow of malloc()

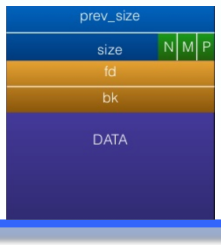
- 第二次以後執行 malloc，只要分配出去的大小不超過 128KB，就不會再執行 system call 跟系統要空間。
- 即使將 main arena 所分配出去的記憶體 free 掉，也不會立即還給 kernel
- Free 的記憶體空間交給 glibc 來管理

Chunk

- **Chunk**
  - glibc 實作記憶體管理時的資料結構
  - malloc() 所分配出去的記憶體空間就是一個 chunk
  - chunk header + allocated user data
  - chunk 被 free 後會被加入 bin 中
- **Allocated Chunk**
- **Free Chunk**
- **Top Chunk**
  - 分配完後剩下的空間



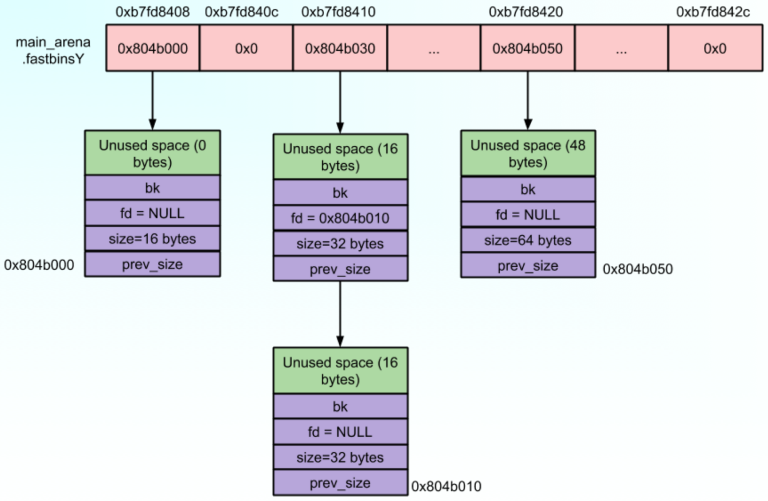
N: Non main arena
P: Previous Inuse?
M: is_mmapped?



bin

- **bin**
  - 一種 linked list
  - 為了讓 malloc 可以更快找到合適大小的 chunk 而設計。在 free 一個 chunk 時，會依據該 chunk 的大小將其加入合適的 bin 中。
- **Fast bin (LIFO): < 64B**
- **Small bin (circular doubly linked list): < 512B**
- **Large bin (circular doubly linked list): >= 512B**
- **Unsorted bin (circular doubly linked list): > 64B**

bin



Fast Bin Snapshot

7
Interrupts

7-1
Exception & Interrupt

Exception

- **Definition**
  - Event that disrupts normal execution and forces processor into execution of special instructions in privileged state
- **Category**
  - Synchronous exceptions and asynchronous exceptions
- **Sync Exception: raised by internal events**
  - Alignment exception - read()/write() that begin at an odd memory address cause a memory access error event and raise an exception
  - Arithmetic operation that results in a division by zero raises an exception.
- **Async Exception: raised by external events from h/w device**
  - System reset exception - Pushing the reset button on the embedded board triggers an asynchronous exception
  - Communications processor module is an external device that can raise asynchronous exceptions when it receives data packets

Interrupt

- **Definition**
  - External interrupt: an asynchronous exception triggered by an event that an external hardware device generates
- **Note**
  - If misused, exception and interrupt can become the source of troubled designs

Applications of Exceptions and Interrupts

- **Exceptions and interrupts help the embedded engineer in:**
  - Internal errors and special conditions management,
  - Hardware concurrency, and
  - Service requests management

Internal Errors and Special Conditions Management

- **Goal**
  - Handling and appropriately recovering from a wide range of errors without coming to a halt
- **Exceptions are either error conditions or special conditions that the processor detects while executing instructions**
  - Error conditions can occur for a variety of reasons
    - Division by zero
    - Over-flow
    - Memory read or write failures
    - Attempts to access floating-point hardware when not installed
  - Special conditions are exceptions generated by special instructions
    - Motorola 68K processor family: TRAP instructions
      - These instructions force processor to move into privileged mode, consequently gaining access to privileged instructions
        - » [Example] trace exception generated by break point feature: debugger agent handles this exception and let host debugger to perform s/w break point and code stepping
- **Two execution modes: normal and privileged**
  - Privileged instructions: can be executed only in privileged mode

Hardware Concurrency

- **Definition**
  - Ability to perform different types of work simultaneously
  - Many external hardware devices can perform device-specific operations in parallel to the core processor
  - These devices require minimum intervention from the core processor
  - The key to concurrency is knowing when the device has completed the work previously issued so that additional jobs can be given
    - External interrupts are used to achieve this goal.
- **Example**
  - Embedded application issues work commands to a device
    - Embedded application continues execution on core processor while the device tries to complete the work issued
    - After the work is complete, the device triggers an external interrupt to the core processor, which indicates that the device is now ready to accept more commands

Service Request Management

- **Definition**
  - Use external interrupts to signal or alert an embedded processor that an external hardware device is requesting service
- **Example**
  - Timer chip communicates with the embedded processor through an interrupt when a preprogrammed time interval has expired
  - Network interface device uses an interrupt to indicate the arrival of packets after the received packets have been stored into memory

Closer Look at Exceptions and Interrupts

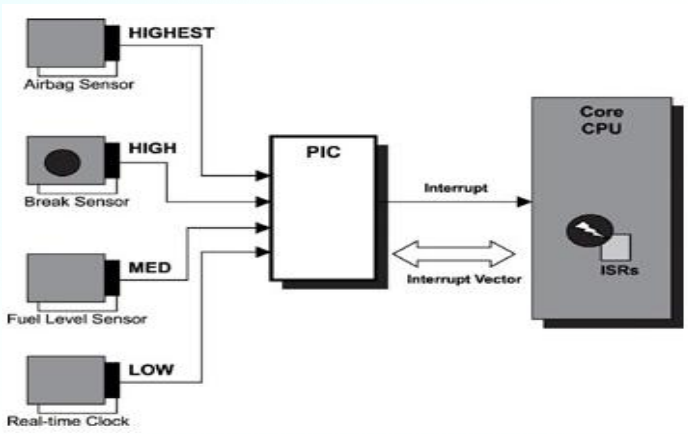
- **Programmable Interrupt Controllers and External Interrupts**
- **Classification of General Exceptions**
- **General Exception Priorities**

PIC and External Interrupts

- **Programmable interrupt controller (PIC)**
  - PIC is implementation-dependent
  - PIC provides two main functionalities:
    - Prioritizing multiple interrupt sources
      - Highest priority interrupt is presented to the core CPU for processing
    - Offloading the core CPU
      - Determine an interrupt's exact source
  - PIC has a set of interrupt request lines, each has a priority assigned to it
    - External source generates interrupts by asserting a physical signal on the interrupt request line

Example of PIC

- **Four interrupt sources**
  - Each source connects to one distinct interrupt request line
    - Airbag sensor
    - Break sensor
    - Fuel-level sensor
      - detecting the amount of gasoline
    - Real-time clock
- **Interrupt Table**
  - Lists all available interrupts in the embedded system



Interrupt Table

Source	Priority	Vector Addr.	IRQ	Max Freq.	Description
Airbag Sensor	Highest	14h	8	N/A	Deploys airbag
Break Sensor	High	18h	7	N/A	Deploys the breaking system
Fuel Level Sensor	Mid	1Bh	6	20Hz	Detects the level of gasoline
Real-Time Clock	Low	1Dh	5	100Hz	Clock runs at 10ms ticks

- **Priority:** higher priority interrupt source can preempt the processing of a lower priority interrupt
- **Max Freq. :** process time constraint placed on all ISRs that have the smallest impact on the overall system
- **Vector Addr.:** memory address that the ISR must be installed
- **IRQ:** Interrupt number

Classification

- **Types of exceptions**
  - Asynchronous-non-maskable
  - Asynchronous-maskable
  - Synchronous-precise
  - Synchronous-imprecise
- **External interrupts are asynchronous exceptions**
  - Can be blocked or enabled by software are called *maskable exceptions*
  - Cannot be blocked by software are called *non-maskable exceptions*
    - Non-maskable exceptions are acknowledged by the processor and processed immediately
    - Hardware-reset exceptions are always non-maskable exceptions
    - Many embedded processors have a dedicated non-maskable interrupt (NMI) request line: device connected to the NMI request line can generate an NMI
- **External interrupts, with the exception of NMIs, are the only async. exceptions that can be disabled by software**

Classification (Cont.)

- **Synchronous exceptions: precise and imprecise exceptions**
  - Precise exception
    - Processor's program counter points to the exact instruction that caused the exception: offending instruction
    - Processor knows where to resume execution upon return
    - Modern architectures with pipelining
      - Exceptions are raised to processor in the order of written instruction, not execution instruction
      - Instructions that follow offending instruction and that were started in pipeline during exception do not affect the CPU state
    - Chip vendors employ advanced techniques to streamline overall execution and increase chip performance
      - Predictive instruction and data loading
      - Instruction and data pipelining
      - Caching mechanisms
      - Example: floating point and integer memory operations can be done out of order with non-sequential memory access mode

Classification (Cont.)

- Imprecise exception
  - Embedded processor implements heavy pipelining or pre-fetch algorithms, and cannot determine the exact instruction and associated data that caused an exception
  - When exceptions occur, the reported program counter does not point to the offending instruction
    - program counter is meaningless to the exception handler
- **Note**
  - Knowing the type of exception helps the programmer determine how the system is to recover from the exception

General Exception Priorities

- **All processors handle exceptions in a defined order**
- **Highest: usually reserved for system resets, significant events, or errors that warrant the overall system to reset**
- **High/Mid: reflect a set of errors and special execution conditions internal to the processor**
  - Synchronous exception is generated and acknowledged only at certain states of the internal processor cycle. Sources of these errors are rooted in either the instructions or data
- **Low: asynchronous exception external to the core processor**
  - External interrupts (except NMIs) are the only exceptions that can be disabled by software

Priority	Type	
Highest	Asynchronous	
High	Synchronous	
Mid	Synchronous	
Lowest	Asynchronous	

Priority Scheme

- **From an application point of view, all exceptions have processing priority over operating system objects, including tasks, queues, and semaphores**

Processing Exceptions

- **Exception handling is similar to interrupt handling**
  - Processor takes steps when an exception/external interrupt is raised:
    - Save the current processor state information
    - Load exception/interrupt handling function into the program counter
    - Transfer control to the handler function and begin execution
    - Restore processor state info after the handler function completes
    - Return from the exception/interrupt and resume previous execution
  - A typical handler function does the following:
    - Switch to an exception frame or an interrupt stack
    - Save additional processor state information
    - Mask current interrupt level but allow higher priority interrupts to occur
    - Perform a minimum amount of work so that a dedicated task can complete the main processing

Installing Exception Handlers

- **Exception/interrupt service routines (ESRs/ISRs) must be installed before handling exceptions/interrupts**
  - Installation requires to know the exception/interrupt table (called general exception table)
  - General exception table has a vector address column: vector table
    - Each vector address points to the beginning of an ESR or ISR
    - Installation requires replacing the vector table entry with the address of the desired ESR or ISR
  - Embedded system startup code typically installs ESRs upon initialization
  - Hardware device drivers typically install ISRs upon driver initialization
- **If no associated handler function is installed, the system suffers a system fault and may halt**
  - Solution: install default handler functions into vector table for possible exception and interrupt
    - perform small amounts of work to ensure the proper reception of and the proper return from exceptions
  - Many RTOSes allow programmer to overwrite default handler function with his own or insert further processing in addition to default actions
    - Programmer can code specific actions before and after the default action is completed

Saving Processor States

- **When exception/interrupt occurs and before invoking ESR/ISR, processor must perform some operations to ensure a proper return of program execution**
- **ESR/ISR need to store some info (*processor state info*) in memory**
  - Processor typically saves a min amount of its state info, including
    - status register (SR): contains current processor execution status bits
    - program counter (PC): contains returning address, which is the instruction to resume execution after the exception
  - ESR/ISR must do more to preserve more complete state info in order to properly resume the program execution that the exception preempted
- **Whose stack is used during the exception and interrupt processing?**

Processor State Info vs. Stack

- **Stacks are used for saving processor state information**
  - A statically reserved block of memory with
  - An active dynamic pointer called a *stack pointer*
- **In Motorola's 68000, 2 separate stacks are used: *user stack* (USP) & *supervisor stack* (SSP)**
  - USP is used when the processor executes in non-privileged mode
  - SSP is used when the processor executes in privileged mode

•Stack Operations

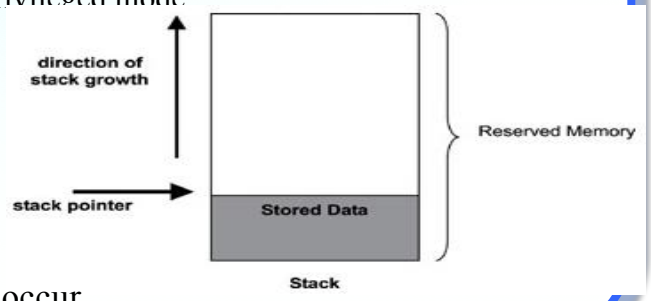
•Push: saving data, SP ++

•Pop: coping off data, SP--

-SP always points to the first valid stack data

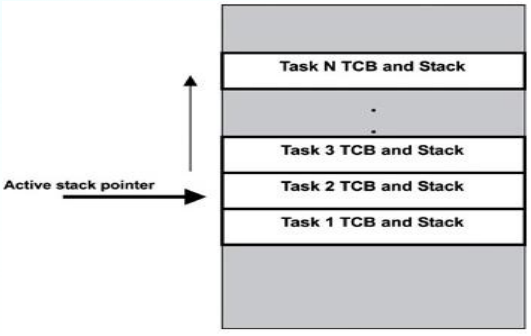
-push/pop items in a symmetric order

-If not, unintended results are likely to occur



Task TCB and Stack

- **In embedded operating system, all task objects have a task control block (TCB)**
  - During creation, a block of memory is reserved as a stack for task use
  - C and C++ typically use the stack space to pass variables between functions and objects
  - Active stack pointer (SP) is reinitialized to that of the active task each time a task context switch occurs: real-time kernel performs this work
  - Processor uses whichever stack the SP points to for storing its min state info before invoking the exception handler



Disable Interrupts

- **Disabling External Interrupt**
  - External interrupt is the only exception type that can be disabled by s/w
  - In many architectures, external interrupts can be disabled or enabled through a processor control register
    - Control register directly controls the operation of the PIC and determines which interrupts the PIC raises to the processor
    - All external interrupts are raised to the PIC
    - PIC filters interrupts according to the setting of the control register and determines the necessary action
- **Interrupt can be disabled, active, or pending**
  - *Disabled interrupt* is also called a *masked interrupt*
    - PIC ignores a disabled interrupt
  - *Pending interrupt* occurs when processor is currently processing a higher priority interrupt
    - Pending interrupt is acknowledged and processed after all higher priority interrupts have been processed
  - *Active interrupt* is the one that the processor is acknowledging and processing

Loading Exception Vector

- **For synchronous exceptions**
  - Processor first determines which exception has occurred
  - Processor calculates the correct index into vector table to retrieve ESR
    - Calculation is dependent on implementation
- **When an asynchronous exception occurs**
  - An extra step is involved: PIC must determine if the interrupt has been disabled (or masked)
    - If masked, PIC ignores interrupt and processor state is not affected
    - If not masked, PIC raises interrupt to processor
      - Processor calculates the interrupt vector address and then loads the vector for execution

Stack Overflow

- **Stack Overflow**
  - When interrupts can nest, the stack must be large enough to hold
    - maximum requirements for the application's own nested function
    - maximum exception or interrupt nesting possible
- **Sample Scenario**
  - Task 2 is currently running and low-priority interrupt is received
  - Task 2 is preempted for a low-priority interrupt
  - Stack grows to handle exception processing storage needs
  - Medium-priority interrupt is received before exception is complete
  - Stack grows to handle medium-priority interrupt processing storage needs
  - High-priority interrupt is received before execution of medium interrupt is complete
  - Stack grows to handle high-priority interrupt processing storage needs

Stack Overflow (Cont.)

- **Without a MMU, no bounds checking is performed when using a stack as a storage medium**
  - In the example, sum of the application/exception stack space requirement is less than the actual stack space allocated by Task 2
  - When data is copied onto the stack past the statically defined limits, Task 3's TCB is corrupted, which is a *stack overflow*
  - Corrupted TCB is not likely to be noticed until Task 3 is scheduled to run
    - These types of errors can be very hard to detect
    - Sometimes, dependably recreating errors is almost impossible
- **Two solutions**
  - Increasing application's stack size to hold all possibilities and the deepest levels of exception and interrupt nesting
  - Having the ESR/ISR switch to its own exception stack, called an *exception frame*

Stack Overflow (Cont.)

- **Max exception stack size is a direct function of**
  - Number of exceptions
  - Number of external devices connected to each distinct IRQ line
  - Priority levels supported by the PIC
- **Simple solution is having the application allocate a large enough stack space to accommodate the worst case**
  - In case if the lowest priority exception handler executes and is preempted by all higher priority exceptions or interrupts
- **A better approach is using an independent exception frame inside the ESR/ISR**
  - Requires far less total memory than increasing every task stack by the necessary amount

Exception Handlers

- **After control is transferred to the exception handler, the ESR or the ISR performs the actual work of exception processing**
  - Usually the exception handler has two parts
    - First part executes in the exception or interrupt context
    - Second half executes in a task context

Exception Frames

- ***Exception frame*: also called the *interrupt stack* in the context of asynchronous exceptions**
- **Why need an exception frame?**
  - To handle nested exceptions
  - As embedded architecture becomes more complex, the ESR/ISR consequently increases in complexity
    - Commonly, exception handlers are written in both machine assembly language and in a high-level programming language
      - The portion of the ESR or ISR written in high-level language requires a stack to pass function parameters
- **Common approach to exception frame is for ESR/ISR to allocate memory (static or dynamic) before installing itself**
  - Exception handler then saves the current stack pointer into temporary memory storage, reinitializes the stack pointer to this private stack, and begins processing

Switching SP to Exception Frame

- **Exception handler can perform more work, such as**
  - storing additional processor state information onto this stack

Differences between ESR and ISR

- **Differences**
  - Exception handler (in many cases) cannot prevent other exceptions from occurring, while an ISR can prevent interrupts of the same or lower priority from occurring
  - Additional processor state information needs to be saved
  - The three ways of masking interrupts are:
    - Disable the device so that it cannot assert additional interrupts
      - Interrupts at all levels can still occur
    - Mask the interrupts of equal or lower priority levels, while allowing higher priority interrupts to occur
      - Device can continue to generate interrupts, but processor ignores them
    - Disable global system-wide interrupt request line to the processor (the line between the PIC and the core processor)
      - Interrupts of any priority level do not reach the processor
        - » Equivalent to masking interrupts of the highest priority level

Differences between ESR and ISR (Cont.)

- **An ISR would typically deploy one of these three methods to disable interrupts for one or all of these reasons:**
  - ISR tries to reduce the total number of interrupts raised by the device
  - ISR is non-reentrant
  - ISR needs to perform some atomic operations
- **Some architectures keep the information on which interrupts or interrupt levels are disabled inside the system status register**
- **Other architectures use an interrupt mask register (IMR)**
  - ISR needs to save the current IMR onto the stack and disable interrupts according to its own requirements by setting new mask values into the IMR
  - IMR only applies to maskable asynchronous exceptions and is not saved by synchronous exception routines

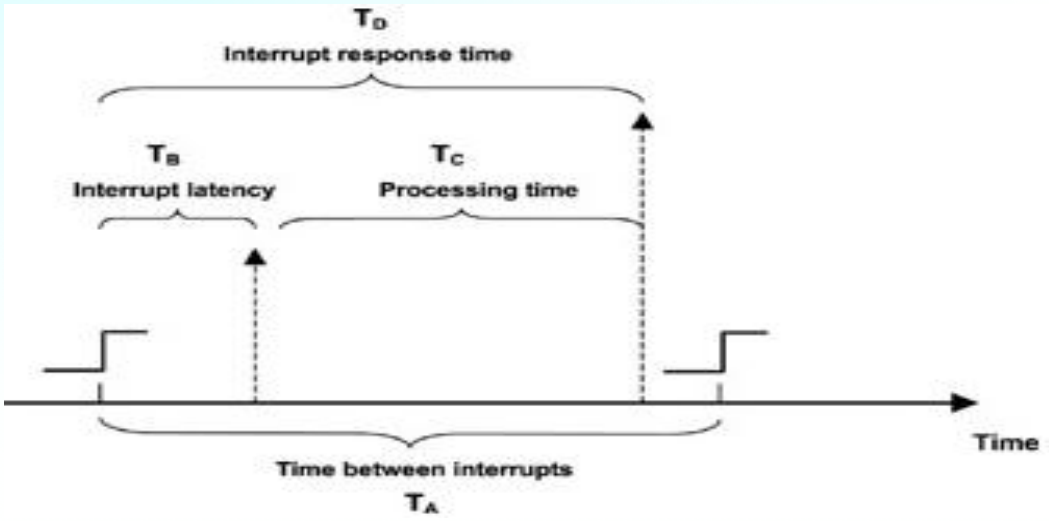
Exception Timing

- **Keep the ESR or ISR short!!**
  - How short should it be?
- **When implementing ISR, designer should know the frequency of each device that can assert an interrupt**
  - Maximum Frequency in the interrupt table: indicates how often a device can assert an interrupt when the device operates at maximum capacity
  - Allowed duration for an ISR to execute with interrupts disabled without affecting the system can be inferred from the interrupt table
  - When executing with interrupts disabled, ISR can cause the system to miss interrupts if the ISR takes too long
  - *Interrupt miss*: an interrupt is asserted but the processor could not record the occurrence due to some busy condition
    - ISR is not invoked for that particular interrupt occurrence
- **Interrupt processing has higher priority than task processing**
  - Real-time tasks that have stringent deadlines can also be affected by a poorly designed ISR

Exception Timing (Cont.)

- **Interrupt latency, T_B: interval between the time when interrupt is raised and the time when ISR begins to execute**
  - Interrupt latency is attributed to:
    - Amount of time it takes the processor to acknowledge the interrupt and perform the initial housekeeping work
      - Always a contributing factor to interrupt latency
    - A higher priority interrupt is active at the time
    - Interrupt is disabled and then later re-enabled by software
  - Interrupt latency can be unbounded and is outside the control of ISR
    - Response time can also be unbounded
    - Processing time TC is determined by how the ISR is implemented
- **Interrupt response time is $T_D = T_B + T_C$.**

Exception Timing (Cont.)



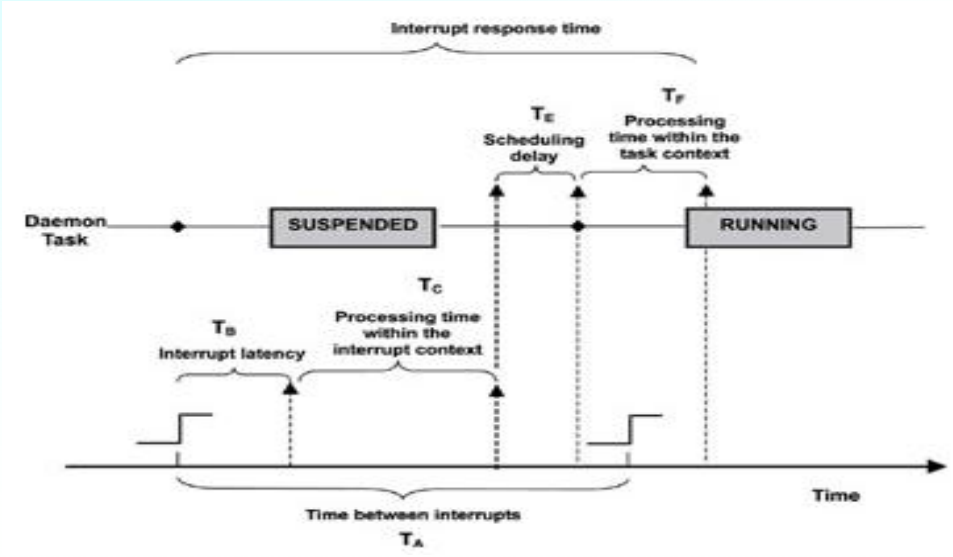
Exception Timing (Cont.)

- **Another approach is to have one section of ISR running in the context of the interrupt and another section running in the context of a task**
  - First section: services the device so that the service request is acknowledged and the device is put into a known operational state so it can resume operation
    - This portion of the ISR packages the device service request and sends it to the remaining section of the ISR that executes within the context of a task
  - Latter part: typically implemented as a dedicated daemon task
- **Why to partition the ISR into two pieces?**
  - To reduce the processing time within the interrupt context
  - Allows the other higher priority task to complete with limited impact

Exception Timing (Cont.)

- **Benefit of partition ISR into two pieces**
  - Lower priority interrupts can be handled with less priority than more critical tasks
  - Reduces the chance of missing interrupts
  - Affords more concurrency because devices are being serviced minimally so that they can continue operations while their previous requests are accumulated without loss to the extent allowed by the system
- **Interrupt response time increases**
  - Interrupt response time is $T_D = T_B + T_C + T_E + T_F$
  - The increase is attributed to the
    - Scheduling delay
      - When other higher priority tasks are running or scheduled to run
      - Includes time needed to perform context switch after daemon task is moved from ready queue to run queue
    - Daemon task might have to yield to higher priority tasks
- **Duration of ISR depends on**
  - #interrupts and frequency of each interrupt source

Exception Timing (Cont.)



General Guides

- **On architectures where interrupt nesting is allowed:**
  - ISR should disable interrupts of the same level if the ISR is non-reentrant.
  - ISR should mask all interrupts if it needs to execute a sequence of code as one atomic operation.
  - ISR should avoid calling non-reentrant functions
    - Some standard library functions are non-reentrant, such as many implementations of malloc and printf
    - Because interrupts can occur in the middle of task execution and because tasks might be in the midst of the "malloc" function call, the resulting behavior can be wrong if the ISR calls this same non-reentrant function
  - ISR must never make any blocking or suspend calls
    - Making such a call might halt the entire system.
- **If an ISR is partitioned into two sections with one section being a daemon task, the daemon task does not have a high priority by default**
  - Priority should be set with respect to the rest of the system

7-2
Signal

Signals

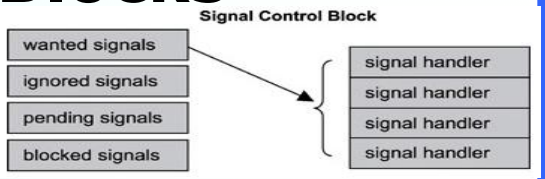
- **Signal: software interrupt, generated when event occurred**
  - Diverts the signal receiver from its normal execution path
  - Triggers the associated asynchronous processing
  - Signals notify tasks of events occurred during execution
  - Signal is associated with an event
    - Unintentional event: illegal instruction
    - Intentional event: notification to task
- **Normal interrupts: events are asynchronous to task and do not occur at any predetermined point in task's execution**
- **Signal vs. Normal interrupts**
  - Signals are so-called software interrupts, which are generated via the execution of some software within the system
  - Normal interrupts are usually generated by the arrival of an interrupt signal on one of the CPU's external pins
    - Not generated by software within the system but by external devices
- **Number/type of signals are system-/RTOS-dependent**

Signals (cont)

- **Task can**
  - Specify particular actions to undertake when signal arrives
- **Task cannot**
  - have control over WHEN it receives signals
    (arrivals are quite random)

- **When a signal arrives,**
  - Task is diverted from its normal execution path
  - Corresponding signal routine is invoked
- **Signal routine, signal handler, asynchronous event handler, and asynchronous signal routine (ASR) are interchangeable**
  - Each signal is identified by an integer value, which is the signal number or vector number.

Signal Control Blocks



- **Signal Control Block is part of TCB**
  - Maintains a set of wanted signals
- **How?**
  - When task is prepared, it said, “the task is ready to catch the signal”
    - Task can
      - provide a signal handler or
      - execute a default handler that the kernel provides
      - process signal first and then pass it on for additional processing by default handler
      - blocks signal from delivery during certain stages when task can not be interrupted
    - It is possible to have a single handler for multiple types of signals
    - Signals can be ignored, made pending, processed (handled), or blocked
  - When signal interrupts task, it said, “the signal is raised to the task”

Signals Sets

- **Wanted signals set**
  - Signals that task is prepared to handle are kept in wanted signals set
- **Ignored signals set**
  - Signals to be ignored by task are maintained in ignored signals set
    - Any signal in ignored signal set does not interrupt the task
- **Pending signals set**
  - Other signals can arrive while task is processing another signal
    - Additional signal arrivals are kept in the pending signals set
      - Signals in pending signal set are raised to the task as soon as the task completes processing the previous signal
      - Pending signals set is a subset of the wanted signals set
- **Blocked signals set**
  - Task can instruct kernel to block certain signals by setting this set
  - Kernel does not deliver any signal from this set until that signal is cleared from the set

Typical Signal Operations

- **Catch**
  - Installs a signal handler
- **Release**
  - Removes a previously installed handler
- **Send**
  - Sends a signal to another task
- **Ignore**
  - Prevents a signal from being delivered
- **Block**
  - Blocks a set of signal from being delivered
- **Unblock**
  - Unblocks the signals so they can be delivered

Catch

- **Task can catch signal after task has specified a handler (ASR) for signal**
- **CATCH operation installs a handler for a particular signal**
  - Kernel interrupts the task's execution upon the arrival of the signal
  - Handler is invoked
  - Task can install the kernel-supplied default handler, the *default actions*, for any signal
  - Task-installed handler has the options of either
    - Processing the signal and returning control to the kernel
    - Processing the signal and passing control to the default handler for additional processing
  - Handling signals is similar to handling hardware interrupts
  - Nature of the ASR is similar to that of the interrupt service routine
- **Signal Vector Table**
  - Element in vector table is pointer or offset to ASR
    - NULL if no handler is assigned to signal
- **Example**
  - After three catch operations have been performed
  - CATCH operation installs one ASR
    - Writing pointer or offset to ASR into table

Release, Send, Ignore, Block, Unblock

- **Handler is invoked if signal is rec'd by any task, not just the task installed it**
- **RELEASE de-installs a signal handler**
  - Task restores the previously installed signal handler after calling release
  - Any task can change the handler installed for a particular signal
    - Saves the previously installed handler before installing its own and
    - Restores handler after it finishes catching the corresponding signal
- **SEND allows one task to send a signal to another task**
- **IGNORE allows task to instruct kernel that never deliver a particular set of signals to that task**
  - If signals cannot be ignored, default handler is used for these signals
- **BLOCK temporarily prevents signals from being delivered**
  - BLOCK protects critical sections of code from interruption
  - BLOCK also prevents conflict when signal handler is already executing and is processing the same signal
  - Signal remains pending while it's blocked.
- **UNBLOCK allows a previously blocked signal to pass**
  - Signal is delivered immediately if it is already pending.

Typical Uses of Signals

- **Some signals are associated w/ h/w events & sent by h/w ISRs**
  - ISR responds to h/w events & send signal to tasks affected by h/w events
- **Signals can also be used for synchronization between tasks**
- **Signals should be used sparingly for the following reasons:**
  - Using signals can be expensive when used for inter-task synchronization
    - Signal alters the execution state of its destination task
    - Since signals occur asynchronously, the receiving task becomes nondeterministic, undesirable in real-time systems
  - Many implementations do not support queuing or counting of signals
    - Multiple occurrences of the same signal overwrite each other
    - Task cannot determine if a signal has arrived multiple times.
  - Many implementations do not support signal to carry information
    - Data cannot be attached to a signal during its generation
  - Many implementations do not support signal delivery order and priority
    - Signal triggered by page fault is more important than task termination
    - On an equal-priority system, the page fault might not be handled first
  - Many implementations do not guarantee when to deliver pending signal
- **Some kernels allow for**
  - Prioritized signal based on signal number
  - Each signal to carry additional information
  - Multiple occurrences of the same signal to be queued

Case study: Asynchronous Event Notification Using Signals

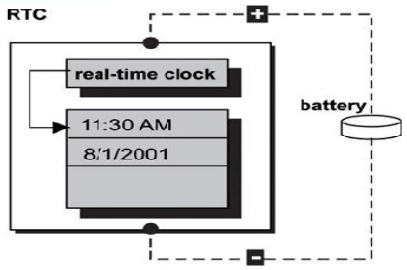
- **One task can synchronize with another task in urgent mode using the signal facility**
  - Signaled task processes the event notification asynchronously
    - Task generates a signal to another task
    - Receiving task diverts from its normal execution path and executes its asynchronous signal routine

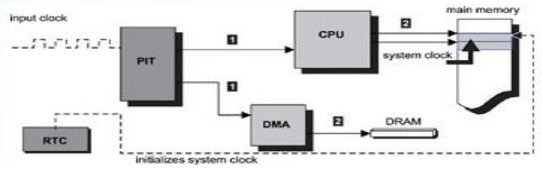
Signal() -Linux

- **sighandler_t signal(int *signum*, sighandler_t *act*);**
- **int sigaction(int *signum*, const struct sigaction **act*, struct sigaction **oldact*);**
  - **signal**() installs a new signal handler for the signal with number signum.
  - The signal handler is set to act which may be a user specified function, or either **SIG_IGN** or **SIG_DFL**
  - Upon arrival of a signal with number signum the following happens.
    - If the handler is set to **SIG_IGN**, then the signal is ignored.
    - If the handler is set to **SIG_DFL**, then the default action associated to the signal occurs
    - If the handler is set to a function act then
      - First either the handler is reset to SIG_DFL or an implementation-dependent blocking of the signal is performed and
      - Next act is called with argument signum
        - » Using a handler function for a signal: "catching the signal“
      - Signals **SIGKILL** and **SIGSTOP** cannot be caught or ignored.
  - **RETURNVALUE**
    - **signal**() returns the value of signal handler, or **SIG_ERR** on error.

7-3
Timer & timer services

Real-Time Clocks and System Clocks





- **Real-time clocks (RTC) track time, date, month, and year**
  - Commonly, clocks are integrated with battery-powered DRAM
- **RTC is independent of CPU & programmable interval timer**
  - Maintenance of real time between system power cycles is possible
- **Job of system clock**
  - Track real-time or elapsed time after system power up
  - Init value is retrieved from RTC at power up or set by user
  - Programmable interval timer drives system clock
    - system clock value increments per timer interrupt
    - Timer interrupt needs to maintain system clock

Programmable Interval Timers

- ***Programmable interval timer* (PIT): *timer chip***
  - Device designed mainly to function as
    - Event counter,
    - Elapsed time indicator,
    - Rate-controllable periodic event generator,
    - Other applications for solving system-timing control problems
  - Commonly incorporated into the embedded processor (*on-chip timer*)
  - Dedicated stand-alone timer chips can reduce processor overhead
- **Functionality of PIT: input clock source**
- **Characteristic of PIT**
  - Fixed frequency
  - A set of programmable timer control registers
- ***Timer interrupt rate*: num of timer interrupts generated per sec**
    - Calculated as a function of the input clock frequency
    - Be set into a timer control register

Timer Countdown Value

- ***Timer countdown value***
  - Determines when the next timer interrupt occurs
  - Be loaded in one of the timer control registers and decremented by one every input clock cycle
  - Remaining timer control registers determine other modes of timer operation
    - Whether periodic timer interrupts are generated
    - Whether countdown value should be automatically reloaded for next timer interrupt
- **Example**
  - If timer chip output pin interconnects with control input pin of DMA chip
    - Timer chip controls the DRAM refresh rate.

Timer Chip Initialization

- **Initialization is performed as part of system startup**
  - Resetting and bringing the timer chip into a known hardware state
  - Calculating proper value to obtain desired timer interrupt frequency and programming this value into appropriate timer control register
  - Programming other timer control registers related to the earlier interrupt frequency with correct values.
    - Timer chip dependent, refer to its hardware reference manual
  - Programming the timer chip with the proper mode of operation
  - Installing the timer interrupt service routine into the system
  - Enabling the timer interrupt

TINTR

- **Behavior of timer chip output is programmable through control registers**
  - *Timer interrupt-rate register* (TINTR) is the most important register
- ***TINTR = F(x)***
  - where *x* = frequency of the input crystal
- **Manufacturers of timer chips provide this function and information is available in programmer's guide**
- **Timer interrupt rate: #(timer interrupt occurrences)/sec**
  - Each interrupt is called a *tick*: unit of time
    - If timer rate is 100 ticks, each tick is an elapsed time of 10 ms
- **Periodic event generation capability of the PIT is important**
  - Heart of RT kernel
    - Announcement of timer interrupt occurrence (*tick announcement*)
      - From ISR to the kernel
      - From ISR to the kernel scheduler, if any
        - » Many kernel schedulers run through their algorithms and conduct task scheduling at each tick.

Timer Interrupt Service Routines

- **Timer ISR performs**
  - Updating the system clock-Both the absolute time and elapsed time is updated.
    - *Absolute time* is time kept in calendar date, hours, minutes, and seconds.
    - *Elapsed time* is usually kept in ticks and indicates how long the system has been running since power up.
  - Calling a registered kernel function (announce_time_tick) to notify the passage of a preprogrammed period
  - Acknowledging the interrupt, reinitializing the necessary timer control register(s), and returning from interrupt

Example of Timer ISR

- **announce_time_tick() is invoked in the context of the ISR**
  - All of the restrictions placed on an ISR are applicable to announce_time_tick()
  - In reality, announce_time_tick() is part of the timer ISR
  - announce_time_tick() is called to notify the kernel scheduler about the occurrence of a timer tick
  - Equally important is the announcement of timer tick to soft-timer handling facility
- **Soft-timer handling facility**
  - maintaining soft timers at each timer tick

Soft-Timer Handling Facility

- **Soft-timer facility include**
  - Allowing applications to start a timer
  - Allowing applications to stop or cancel a previously installed timer
  - Internally maintaining the application timers
- **Soft-timer facility is comprised of two components**
  - One component lives within the timer tick ISR (first phase)
  - The other component lives in the context of a task (second phase)
- **Why two components?**
  - If all of the soft-timer processing is done with the ISR and if the work spans multiple ticks
    - System clock might appear to drift
    - Worse, the timer tick events might be lost
  - Timer tick handler must be short and must be conducting the least amount of work possible
  - Processing of expired soft timers is delayed into a dedicated task because applications using soft timers can tolerate a bounded timer inaccuracy
    - *Bounded timer inaccuracy*: the imprecision the timer may take on any value and the value is guaranteed to be within a specific range

Soft-Timer Handling Facility (Cont.)

- **Workable model**
  - Create a dedicated processing task (worker task) in conjunction with its counter part that is part of the system timer ISR
    - ISR counterpart is given a name of ISR_timeout_fn() for discussion
  - System timer chip is programmed with a particular interrupt rate
  - Associated timer tick granularity is typically much smaller than the granularity required by the application-level soft timers
  - ISR_timeout_fn() must work with this value and notify the worker task

Model of Soft-Timer

- **Assumption**
  - Application requires three soft timers
    - Timeout values equal 200ms, 300ms, and 500ms
      - Least common denominator is 100ms
    - If each h/w timer tick is 10ms, then 100ms→countdown value of 10
- **Concept**
  - ISR_timeout_fn() keeps track of this countdown value and decrements it by one during each invocation
  - ISR_timeout_fn() notifies the worker task by a "give" operation on worker task's semaphore after countdown value reaches zero, effectively allowing task to be scheduled for execution
  - ISR_timeout_fn() then reinitializes countdown value back to 10

Model of Soft-Timer (Cont.)

- **Worker task must maintain an application-level, timer-countdown table based on 100ms granularity**
  - Timer table has three countdown values: 2, 3, and 5
- **An application-installed, timer-expiration function is associated with each timer**
- **3 timers are decremented by worker task each time it runs**
  - When the counter reaches zero, the application timer has expired
    - 200ms timer and its function App_timeout_fn_1(), which application installs, is invoked
    - Single ISR-level timer drives 3 application timers at the task-level
    - These application-installed timers are called *soft timers*
      - Timer processing is not synchronized with the hardware timer tick