CSE 237B Fall 2009
Real Time Operating Systems

Rajesh Gupta
Computer Science and Engineering
University of California, San Diego.
Overview

The goal of task modeling and management is to understand the requirements of embedded software for application programming as well as for the operating system needs.

- Task management
- Task interaction

- Embedded Software as Tasks
  - Static and Dynamic Aspects of Task scheduling

- Memory Management: Stack and Heap
- Real-time kernels
- Commercial and research real-time operating systems
Memory Management

• Role of Stack and Heap in runtime system
  – Static versus dynamic memory use estimation
  – Predictable versus controlled memory usage

• General approaches to memory management
  – Stop the world
  – Do it incrementally
  – Do it incrementally and consistently
Memory Management

• Two issues
  – Heap management
  – Stack management

• Heap management
  – Classic heap
  – Priority heap
  – Fixed block heap

• Classic heap
  – Usually found on Unix systems
  – The memory is collected into one giant heap and partitioned according to the demand from tasks.
  – There are several “fit” memory allocation algorithms, e.g., best-fit, first-fit, that also attempt to minimize the memory fragmentation.
  – Has a big management overhead so is not used in real-time systems (STW)
Heap Management

• Priority heap
  – partitions the memory along priority boundaries, e.g., a high and a low priority partitions are created
  – the high priority partition is given adequate memory for its worst case, and the remaining memory is given to the low priority partition.
  – In this scheme, only low priority tasks may need to wait due to insufficient memory.

• Fixed block heap
  – partitions the memory into several pools of fixed block length and upon a request, allocates a single block of memory from the pool with size equal or larger than the requested amount.
  – Partitions can be used to keep multiple tasks in memory at the same time, provided that the number of tasks is fixed and known.
  – leads to fragmentation. One approach to minimize this is that memory can be divided into regions in which each region contains a collection of different-sized, fixed-sized partitions.
Other Heap Issues

• Debug information:
  – Saving information with each block of memory can help characterize memory usage and performance. The additional information is useful in isolating such memory problems as memory leaks, over-committed memory, and thrashing.

• Keeping additional information in the memory blocks allocated:
  – A memory block can be stamped with the id of the task (task id) or the address of the return address of the routine that requests it.
    • These can help identify the causes of the memory problems.
  – Time stamping in conjunction with task id or return address makes it possible to see how long a memory block has been allocated; if the time is long, it is usually a sign that the memory block was lost.
  – During the development phase, it may be desirable to have a periodic task running that examines the time stamps to identify the blocks with potential memory leaks.
Stack Management

• Stack management:
  – When multiple tasks share a single processor, their contexts (volatile information such as the contents of hardware registers, memory-management registers, and the program counter) need to be saved and restored so as to switch them.
  – This can be done using task-control block model OR one or more run-time stacks

• Task-control block model
  – best for full-featured real-time operating systems
  – Context is kept in the control block of the task.
  – Having multiple tasks means multiple control blocks, which are maintained in a list.
    • This list can be fixed, if the number of tasks is fixed and known, or dynamic.
    • In the fixed case, the control block of a task is regarded as “deleted” even though it is not. In the dynamic case, it is actually deleted and the memory allocated for the task is returned to the heap.
Stack Management

• Run-time stacks
  – used to keep context
  – may use only one run-time stack for all the tasks or one run-time stack in conjunction with several application stacks (or private stacks), one for each task in memory
  – Multiple stack case allows tasks to interrupt themselves, thus helping handle transient overload conditions, or reentrancy and recursion.
  – Stack size must be known a priori. If recursion is not used, the stack size is equal to the number of tasks anticipated plus some provision for time overloading.
  – Operating system manages the stacks.
  – If it is necessary to maintain the context of an ISR over repeated interrupts, then an independent stack rather than the stack that is used for all the interrupts is required.
Real-Time Kernel Issues

- **Scheduler**: the component that determines which task to run next.
- **Dispatcher**: the component that gives control of the processor to the task selected by the scheduler.
- **Kernel**: the smallest portion of the operating system that provides
  - task scheduling, dispatching, and intertask communication.
  - should be as fast as possible as it is invoked during every task switch
  - can be implemented in hardware or software
- **Kernel types**
  - **Nanokernel** - the dispatcher
  - **Microkernel** - a nanokernel with task scheduling
  - **Kernel** - a microkernel with intertask synchronization
  - **Executive** - a kernel that includes privatized memory blocks, I/O services, and other complex issues. Most commercial real-time kernels are in this category.
  - **Operating system** - an executive that also provides generalized user interface, security, file management system, etc.
Real-Time Kernel Issues

• Kernel design strategies
  – Polled loop systems
  – Coroutines
  – Interrupt-driven systems
  – Foreground/background systems

• Foreground/background systems
  – most common solution for embedded applications
  – involve a set of interrupt driven tasks called the foreground and a collection of noninterrupt driven tasks called background
  – run the foreground task in a round-robin, preemptive priority, or a combination of both
  – run the background tasks in a mode that allows any foreground task to preempt them
  – This scheme is similar to the model with periodic and aperiodic tasks in task scheduling theory.
Real-Time Operating Systems

• Three groups
  – Small, fast, proprietary kernels
  – Real-time extensions to commercial operating systems
  – Research operating systems

• Small, fast, proprietary kernels
  – homegrown
  – commercial offerings
    • QNX, PDOS, pSOS, VCOS, VRTX32, VxWorks, ERCOS
  – To reduce the run-time overheads incurred by the kernel and to make the system fast, the kernel
    • has a fast context switch
    • has a small size
    • responds to external interrupts quickly
    • minimizes intervals during which interrupts are disabled
    • provides fixed or variable sized partitions for memory management as well as the ability to lock code and data in memory
    • provides special sequential files that can accumulate data at a fast rate
RTOS: Proprietary Kernels

- To deal with timing constraints, the kernel
  - provides bounded execution time for most primitives
  - maintains a real-time clock
  - provides for special alarms and timeouts
  - supports real-time queuing disciplines such as earliest deadline first and primitives for jamming a message into the front of a queue
  - provides primitives to delay processing by a fixed amount of time and to suspend/resume execution

- Also, the kernel
  - performs multitasking and intertask communication and synchronization via standard primitives such as mailboxes, events, signals, and semaphores.

- For complex embedded systems, these kernels are inadequate as they are designed to be fast rather than to be predictable in every aspect.
Example: ERCOS

• Embedded Real-Time Control Operating System
  – targeted for automotive applications, circa 1996
  – E.g., engine control, transmission control
• A typical engine controller
  – Memory: 256 KB ROM, 32 KB RAM
  – interfaces to about 80 sensors, actuators
  – connected to a real-time communication network, e.g., CAN bus
  – software:
    • about 100 concurrently executing tasks
    • most demanding task is injection control, must be precise within a few microseconds
Tasks

Time Triggered Systems
- Non-preemptive S-task
  - Two states: Inactive, Active
- Preemptive S-task
  - States: Inactive, Active: Ready, Active: Running
  - An active task can be preempted from Running to Ready

Event Triggered Systems
- Non-preemptive S-task
- Preemptive S-task
- C-task
  - In addition to Ready and Running, C-task can be Blocked
  - States: Inactive, Active:Ready, Active:Running, Active:Blocked
Tasks in ERCOS

• S-tasks grouped into schedule sequences
• These sequences are static schedules (time-triggered) that are developed off-line
  – therefore, task dependencies (mutex, precedence) are built into the schedule, no explicit synchronization is needed
• Dynamic scheduling based on task priorities
  – cooperative (non-preemptive at the task level) scheduling,
    • a context switch may only take place between the tasks of a schedule sequence
    • allows a critical section to be completely encapsulated within a task, simplified data consistency model
  – or, preemptive scheduling
    • needed for short response time and minimal jitter
    • to avoid blocking, uses a priority ceiling protocol
      – when a task get a resource, it raises its priority to the ceiling of the resource. So it is not possible to preempt a task that holds a needed resource.
Interprocess Communication in ERCOS

• Done by state messages

• Task activation
  – copy input messages into a local data structure

• Task completion
  – copy it (output data structure) back into the global data area

• Optimizations
  – inline expansion of send/receive operations as assignment operations
  – batching of send/receive operations to reduce the number of message copies in a schedule sequence
Error Detection in ERCOS

- Mechanisms for runtime detection of errors
  - a deadline checking service by the OS to detect late system responses
  - interrupts are monitored continuously after acking an interrupt, the line is disabled for duration of the minimum interarrival period
  - maximum number of total tasks is determined offline and the actual number of tasks is monitored by the OS
  - a watchdog process generates regular life-line messages for the external observers.
Commercial Real-Time Operating Systems

• Real-time extensions to commercial operating systems
  – Unix to RT-Unix
  – POSIX to RT-POSIX
  – MACH to RT-MACH
  – CHORUS to its real-time version

• Properties
  – generally slower and less predictable than the proprietary kernels but have greater functionality and better software development environments
  – based on a set of familiar interfaces that facilitate portability
  – not the correct approach because too many basic and inappropriate underlying assumptions still exist such as optimizing for the average case (rather than worst case), assigning resources on demand, ignoring most if not all semantic information about the application, and independent CPU scheduling and resource allocation possibly causing unbounded blocking.
Implementation in Real-time OSs

- Small, fast, proprietary kernels (commercial, home grown)
- Real-time extensions to commercial OSs
- Research OSs
- Part of language run-time environments
  - Ada
  - Java (embedded real-time Java)
- Monolithic kernel vs. Microkernel
Monolithic Systems

- The OS as a whole runs in privileged mode
- Internal structure applied to invocation of OS services by applications
- Interrupt handling done by the kernel,
  - usually not full-fledged processes, can not invoke most system services
- Interrupt handling blocks OS scheduler
  - Keep overhead small through these restrictions

- One can further reduce processor overhead by running the application as whole in a privileged mode
  - That is, application code is bound to the OS at link time and system calls become regular function calls
  - Faster OS/App interface, but harder debugging, reliability issues, upgrading is difficult
Microkernel Systems

- Basic idea is to move many OS functions from the kernel up into the OS server processes (up in the hierarchy) in the user mode
  - Reducing to minimal code to run in privileged mode
- Interrupts do minimal amount of privileged work and then use communications interface (message passing) to an interrupt service ‘task’ (running in the user mode)
- Main motivation is to structure the communication between application and OS services
  - Enforce security policies on such communications
  - Isolate these from critical OS kernel functions (device IO)
  - Improved reliability as OS services failure does not lead to OS failure
  - Extensible to distributed systems
Comparison of Various RTOS Organizations: Cyclic Executive

Kernel Mode

- Application
- Device Drivers
- I/O Services
- TCP/IP Stack
- Network Drivers
- Hardware
Comparison of Various RTOS Organizations: Monolithic Kernel

User Mode (protected)

Kernel Mode

- Filesystems
- Device Drivers
- Network Drivers
- I/O Managers
- Graphics Drivers
- Graphics Subsystem
- Other…

Hardware Interface Layer

Hardware
Comparison of Various RTOS Organizations: Microkernel

User Mode (protected)

Kernel Mode

Kernel (tiny)
Some Examples

• For tiny systems
  – PALOS – UCLA:
    http://nesl.ee.ucla.edu/projects/ahlos/software/libraries/palos030.tar
  – TinyOS – UCB:

• For mid-size systems
  – µCOS-II
  – eCos

• For large-size systems
  – VxWorks
  – Real-time Linuxes
Example I: PALOS

- Based on [Melkonian00]
- Structure – PALOS Core, Drivers, Managers, and user defined Tasks
- PALOS Core
  - Task control: slowing, stopping, resuming
  - Periodic and aperiodic handlers
  - Inter-task Communication via event queues
  - Event-driven tasks: task routine processes events stored in event queues
    
    ```c
    while (eventQ != isEmpty){dequeue event; process event;}
    ```
- Drivers
  - Processor-specific: UART, SPI, Timers..
  - Platform-specific: Radio, LEDs, Sensors
Sizes

• Thin Core:
  – Code size: 956 bytes; Memory: 548 bytes
  – Processor independent, provide means of managing event queues and exchanging events between tasks
  – Provides means of task execution control (slowing, stopping, resuming)
  – Supports a scheduler: periodic, aperiodic functions.

• ATmega128 processor:
  – Flash (code): 128 Kbytes
  – RAM (memory): 4Kbytes

• Typical usage
  – 3 drivers, 3 user tasks
    • Code size: 8Kbytes
    • Memory: 1.3 Kbytes
Tasks in PALOS

- A task belongs to the PALOS main control loop
- Each task has an entry in PALOS task table (along with eventQs)

**Execution control**

- A task counter is associated with each task
- Counters are initialized to predefined values
  - 0: normal
  - Large positive: slowdown
  - -1: stop
  - non-negative: restart
- Counters are decremented 1) every main control loop iteration (relative timing) 2) by timer interrupts (exact timing)
- When counter reaches zero, the task routine is called. The counter is reset to reload value.
Event Handlers in PALOS

- Periodic or aperiodic events can be scheduled using Delta Q and Timer Interrupt
- When event expires appropriate event handler is called
### Event Driven Task

- **Typical Task Routine**
  ```
  While (eventQ != isEmpty) {
    dequeue event;
    process event;
  }
  ```

- **Task Table**
  ```
  typedef struct {
    SHORT (*initHandler)(void);
    SHORT (*taskHandler)(void);
    SHORT execCounter; /* Counter to be used for task speed control */
    /* when counter reaches zero the task is executed */
    SHORT reloadCounter; /* execCouter is reset the reload counter value after it */
    /* reaches zero */
    SHORT maxEvent; /* stores max number of events that can be processed per */
    /* iteration. can be used to give priority */
    BOOL isExactTiming; /* indicates whether the counter is decremented */
    /* following exact timing */
    USHORT header; /* header ptr */
    USHORT trailer; /* trailer ptr */
    USHORT eventStrSize; /* member structure size */
    USHORT maxQsize; /* max number queue size */
    USHORT curQsize; /* current queue size */
    CHAR isValid; /* indicates whether this is valid entry */
  } eventQ;
  ```
Main Control Loop

// main loop
while (1){ // run each task in order
    for (i=0; i< globalTaskID; i++){
        isExact = qArray[i].isExactTiming;
        tmpCntr=qArray[i].execCounter;
        if ( tmpCntr != TASK_DISABLED) {/* task is not disabled */
            if ( tmpCntr ) {/* counter hasn't expired */
                if (!isExact)
                    qArray[i].execCounter--;
            }
            else {/* exec counter expired */
                if (isExact)
                    PALOSSCHED_TIMER_INTR_DISABLE;
                qArray[i].execCounter = qArray[i].reloadCounter;
                if (isExact)
                    PALOSSCHED_TIMER_INTR_ENABLE;
            } /* run the task routine */
            (*qArray[i].taskHandler)();
        }
    }
}
PALOS Core functions

SHORT palosEvent_register(SHORT (*initFunc)(void), SHORT (*taskFunc)(void), LONG xCounter, LONG rCounter, USHORT maxEv, BOOL exactTiming, USHORT eventStrSize, USHORT maxQsize, void *ev);
SHORT palosEvent_putEvent( USHORT taskID, void *ev, CHAR isAtomic);
SHORT palosEvent_getEvent( USHORT taskID, void *ev, CHAR isAtomic);
SHORT palosEvent_start(USHORT taskID, LONG excCntr, LONG reldCntr);
SHORT palosEvent_stop(USHORT taskID);
SHORT palosEvent_maxEvent(USHORT taskID, USHORT maxEv);
SHORT palosEvent_exactTiming(USHORT taskID, BOOL exactTiming);
SHORT palosSched_schedule( USHORT tid, ULONG param, hndlrWrapper *tmrHandler, ULONG ticks, CHAR isPeriodic);
SHORT palosSched_cancel( USHORT tid, hndlrWrapper *tmrHandler );
PALOS Features

• Portable
  – CPUs: ATmega103, ATmega128L, TMS320, STStrongThumb
  – Boards: MICA, iBadge, MK2

• Small Footprints
  – Core (compiled for ATMega128L)
    • Code Size: 956 Bytes, Mem Size: 548 Bytes
  – Typical (3 drivers, 3 user tasks)
    • Code Size: 8 Kbytes, Mem Size: 1.3 Kbytes

• Task execution control
  – Provides means of controlling task execution (slowing, stopping, and resuming)

• Scheduler: Multiple periodic, and aperiodic functions can be scheduled

• Preliminary version v0.11 available from sourceforge
  https://sourceforge.net/project/showfiles.php?group_id=61125
void UART0_Init();
void UART0_Enable();
void UART0_Disable();
UCHAR UART0_NewData();
UCHAR UART0_GetByte();
USHORT UART0_Get2Bytes();
UCHAR UART0_GetNBytes( UCHAR * ptr_ch, UCHAR nN );
USHORT UART0_Check2Bytes();
UCHAR UART0_GetError();
UCHAR UART0_FreeSpace();
BOOL UART0_WriteByte( UCHAR ch );
BOOL UART0_Write2Bytes( USHORT sh );
BOOL UART0_WriteNBytes( UCHAR * ptr_ch, UCHAR nN );
Task Implementation with PALOS

Need to define event structure
Implement initialization function
Implement main task function
Implement initTask()
  • Performs system initialization
  • Registers different task to PALOS core
Implement initSched()
  • Initial scheduling of events
void main(void)
{
    SHORT i;
    USHORT tmpCntr;
    BOOL isExact;
    // event handler initialization
    palosEvent_init();
    // The user's task is registered and
    // scheduled by this function

    initTask();
    // initialize each function
    for (i=0; i< globalTaskID; i++){
        (*qArray[i].initHandler)();
    }
    // User needs to define this function to
    // schedule events

    initSched();
    // main loop
    while (1){ // run each task in order
        for (i=0; i< globalTaskID; i++){
            isExact = qArray[i].isExactTiming;
            tmpCntr=qArray[i].execCounter;
            if ( tmpCntr != TASK_DISABLED) {
                /* task is not disabled */
                if ( tmpCntr ) {
                    /* counter hasn't expired */
                    if (!isExact)
                        qArray[i].execCounter--;
                }
                else { /* exec counter expired */
                    if (isExact)
                        PALOSSCHED_TIMER_INTR_DISABLE;
                    qArray[i].execCounter =
                        qArray[i].reloadCounter;
                    if (isExact)
                        PALOSSCHED_TIMER_INTR_ENABLE;
                    /* run the task routine */
                    (*qArray[i].taskHandler)();
                }
            }
        }
    }
    /* should never get here */
    return;
}
Example Application for MICA node

StringIn Task: gets string from stdin
StringOut Task: outputs string to stdout
Menu Task: runs the menu state machine to control the frequency of LED flashing frequency

<table>
<thead>
<tr>
<th>PALOS Core</th>
<th>StringIn Task</th>
<th>StringOut Task</th>
<th>Menu Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer1 Driver</td>
<td>UART0 Driver</td>
<td>LED Driver</td>
<td></td>
</tr>
</tbody>
</table>
/ * stringOut task event structure */
typedef struct {
    UCHAR *str;    /* pointer to string */
    UCHAR size;    /* size of the string */
    USHORT eventID;
    ULONG eventParam;
    hndlrWrapper stringOut_TXdone; /* handler called when tx is done */
} stringOut_Event;
SHORT stringOut_init() {
    stringOut_hoqValid=false;

    // stringOut initialization
    UART0_Init();
    return 0;
}

SHORT stringOut_task() {
    UCHAR availSize;

    while ((stringOut_hoqValid == true) ||
        (palosEvent_getEvent(stringOutID,
            &stringOut_hoq, TASK_NON_ATOMIC)
        != PALOSEVENT_QEMPTY)) {
        availSize = UART0_FreeSpace();
        if ( stringOut_hoq.size <= availSize ) {
            UART0_WriteNBytes(stringOut_hoq.str,
                stringOut_hoq.size);
            HANDLER_CALL(stringOut_hoq.stringOut_TXdone,
                stringOut_hoq.eventID,
                stringOut_hoq.eventParam);
            stringOut_hoqValid=false;
        } else {
            UART0_WriteNBytes(stringOut_hoq.str, availSize);
            stringOut_hoq.str += availSize;
            stringOut_hoq.size -= availSize;
            stringOut_hoqValid=true;
            break;
        }
    }
    return PALOSEVENT_TASK_DONE;
}
void initTask() {
SYS_Init();

stringOutID=palosEvent_register(stringOut_init, stringOut_task,
STRINGOUT_DEF_CNTR, STRINGOUT_DEF_RCNTR,
STRINGOUT_DEF_MAXEVENT, false,
sizeof(stringOut_Event), STRINGOUT_Q_SIZE,
(void *)stringOutEvent);

stringInID=palosEvent_register(stringIn_init, stringIn_task,
STRINGIN_DEF_CNTR, STRINGIN_DEF_RCNTR,
STRINGIN_DEF_MAXEVENT, false,
sizeof(stringIn_Event), STRINGIN_Q_SIZE,
(void *)stringInEvent);

menuID=palosEvent_register(menu_init, menu_task,
MENU_DEF_CNTR, MENU_DEF_RCNTR,
MENU_DEF_MAXEVENT, false,
sizeof(menu_Event), MENU_Q_SIZE,
(void *)menuEvent);

palosSchedID=palosEvent_register(palosSched_init, palosSched_task,
PALOSSCHED_DEF_CNTR, PALOSSCHED_DEF_RCNTR,
PALOSSCHED_DEF_MAXEVENT, false,
sizeof(palosSched_Event), PALOSSCHED_EVENTQ_SIZE,
(void *)tEvent);
}
void initSched() {
    // schedule an event
    stringOut_msg(initMsg, MENU_START, 0, HANDLER_WRAP(menu_handler));
}

stringOutTask.c :
void stringOut_msg(CHAR *str, USHORT id, ULONG param, hndlrWrapper *hnd) {
    stringOut_Event outgoingMsgEvent;
    outgoingMsgEvent.str=str;
    outgoingMsgEvent.size=strLength(str);
    outgoingMsgEvent.eventID=id;
    outgoingMsgEvent.eventParam=param;
    HANDLER_COPY(&(outgoingMsgEvent.stringOut_TXdone), hnd);
    palosEvent_putEvent(stringOutID, &outgoingMsgEvent, TASK_NON_ATOMIC);
}
Example II: TinyOS

- OS for UC Berkeley’s Motes wireless sensor nodes
- System composed of
  - Tiny scheduler
  - Graph of components
  - Single execution context
- Component model
  - Basically FSMs
  - Four interrelated parts of implementation
    - Encapsulated fixed-size frame (storage)
      - A set of command handlers
      - A set of event handlers
      - A bundle of simple tasks (computation)
  - Modular interface
    - Commands it uses and accepts
    - Events it signals and handles
- Tasks, commands, and event handlers
  - Execute in context of the frame & operate on its state
  - Commands are non-blocking requests to lower level components
  - Event handlers deal with hardware events
  - Tasks perform primary work, but can be preempted by events
- Scheduling and storage model
  - Shared stack, static frames
  - Events preempt tasks, tasks do not
  - Events can signal events or call commands
  - Commands don’t signal events
  - Either can post tasks
TinyOS Key Claims/Facts

• Stylized programming model with extensive static information
  – Compile time memory allocation
• Easy migration across h/w -s/w boundary
• Small Software Footprint
  – 3.4 KB
• Two level scheduling structure
  – Preemptive scheduling of event handlers
  – Non-preemptive FIFO scheduling of tasks
  – Bounded size scheduling data structure
  – Power-aware: puts processor to sleep when tasks are complete
• Rich and Efficient Concurrency Support
  – Events propagate across many components
  – Tasks provide internal concurrency
  – At peak load 50% CPU sleep
• Power Consumption on Rene Platform
  – Transmission Cost: 1 µJ/bit
  – Inactive State: 5 µA
  – Peak Load: 20 mA
• Efficient Modularity
  – Events propagate through stack <40 µS
• Programming in C with lots of macros, now moving to NestC
• http://webs.cs.berkeley.edu
Example of a TinyOS Component

/* Messaging Component Declarations */

//ACCEPTS:
char TOS_COMMAND(AM_send_msg)(int addr, int type, char* data);
void TOS_COMMAND(AM_power)(char mode);
char TOS_COMMAND(AM_init)();

//SIGNALS:
char AM_msg_rec(int type, char* data);
char AM_msg_send_done(char success);

//HANDLES:
char AM_TX_packet_done(char success);
char AM_RX_packet_done(char* packet);

//USES:
char TOS_COMMAND(AM_SUB_TX_packet)(char* data);
void TOS_COMMAND(AM_SUB_power)(char mode);
char TOS_COMMAND(AM_SUB_init)();
Example of an Event Handler

Start

Bit_Arrival_Event_Handler
State: \{bit\_cnt\}

bit\_cnt++

bit\_cnt==8

Yes

Send Byte Event
bit\_cnt = 0

No

Done
Complete TinyOS Application

Ref: from Hill, Szewczyk et. al., ASPLOS 2000
Example III: µCOS-II

- Portable, ROMable, scalable, pre-emptive, multitasking RTOS
  - Up to 63 statically declared tasks
- Services
  - Semaphores, event flags, mailboxes, message queues, task management, fixed-size memory block management, time management
- Source freely available for academic non-commercial usage for many platforms
  - [http://www.ucos-ii.com](http://www.ucos-ii.com)
  - Value added products such as GUI, TCP/IP stack etc.
  - Book “MicroC/OS-II: The Real-Time Kernel” describes the internals
Example IV: eCos

- Embedded, Configurable OS
- Open-source, from RedHat
- Designed for devices lower-end than embedded Linux
- Several scheduling options
  - bit-map scheduler, lottery scheduler, multi-level scheduler
- Three-level processing
  - Hardware interrupt (ISR), software interrupt (DSR), threads
- Inter-thread communication
  - Mutex, semaphores, condition variables, flags, message box
- Portable
  - Hardware Abstraction Layer (HAL)
- Based on configurable components
  - Package based configuration tool
  - Kernel size from 32 KB to 32 MB
  - Implements ITRON standard for embedded systems
  - OS-neutral POSIX compliant EL/IX API
  - Does not support processes with independent address space even when MMU is available
Example V: Real-time Linuxes

• Several derivatives: uClinux, RTlinux, RTAI
• Microcontroller (no MMU) OSes:
  – uClinux - small-footprint Linux (< 512KB kernel) with full TCP/IP
  – Runs on uC without MMU
  – But has no real-time capability because it inherits processor scheduler from Linux
• QoS extensions for desktop:
  – Linux-SRT and QLinux
    • soft real-time kernel extension
    • target: media applications
• Embedded PC
  – RTLinux, RTAI
    • Similar, architecture outlined in 1997
    • hard real time OS
      – E.g. RTLinux has Linux kernel as the lowest priority task in a RTOS
    • fully compatible with GNU/Linux
  – Interrupt requests by the Linux kernel are not passed to the HW
    • Instead, emulated in the real-time kernel
    • The kernel manages queues and delivery of interrupts