Embedded Operating Systems

Chapters 5 & 9

OS Components

- Kernel mode vs. user mode
- Memory management
  - Logic vs. physical memory
  - Memory allocation
  - Cache management
- Process management
  - Multi-process / multi-task scheduling
  - Interrupt and error handling
- I/O system management
  - Shared I/O devices

CPU Mode

- CPU mode
  - A set of restrictions determine the resources that the instructions can access.
  - x86 CPUs have 4 privilege levels.
- Kernel mode
  - Privilege level 0 : a program can do anything with the system
- User mode
  - Privilege level 3

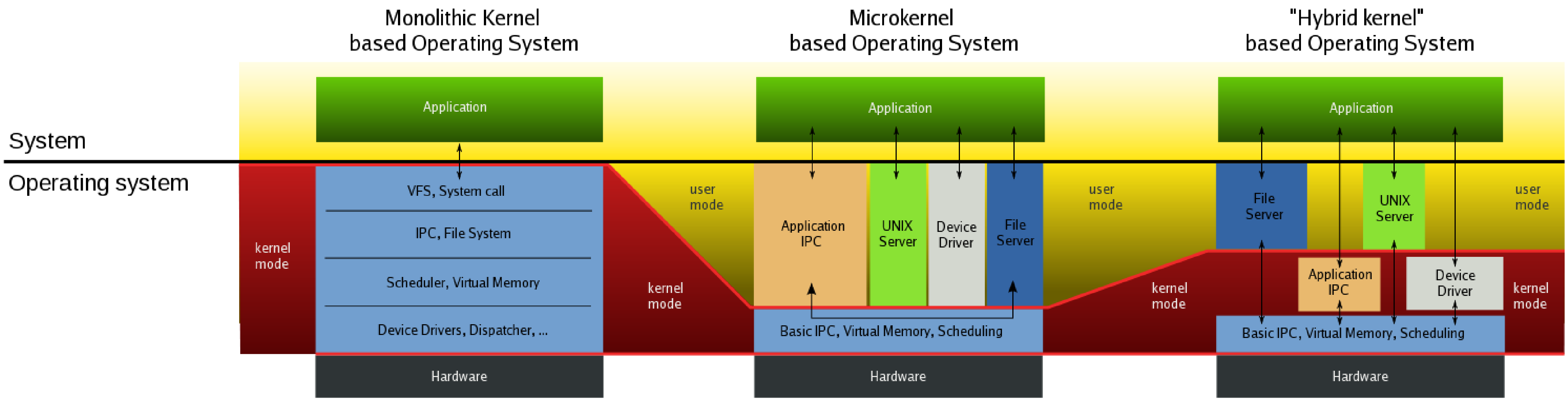
OS Kernel

- A user program has no privilege to directly access any resource.
- The failure of a user program do not impact the rest of the computer system.
- Mode switch
  - When a user program needs to access a resource, the program should get permission from the OS kernel following the required procedure.
  - Then, the OS switches from the user mode to the kernel mode.
  - Example, system calls (device access).

OS Kernel Model

- Monolithic
  - All components are integrated into the kernel.
  - All components provide interfaces to others.
- Layered
  - All components are organized in layers.
  - A component only provides interfaces to its adjacent upper layer.
- Microkernel
  - The kernel is stripped off to include the minimum necessary functions: memory management, process management, and inter-process communication.
  - All other components provide OS services in user space.

OS Kernel Model



OS Components

- Kernel mode vs. user mode
- Memory management
  - Logic vs. physical memory
  - Memory allocation
  - Cache management
- Process management
  - Multi-process / multi-task scheduling
  - Interrupt and error handling
- I/O system management
  - Shared I/O devices

Memory Management

- Manage the mapping between logical (physical) memory and task memory references
- Determine which tasks to load into the available memory space.
- Allocate and deallocate of memory for kernel tasks.
- Support memory allocation and deallocation of application tasks.
- Track the memory usage of system components.
- Ensure cache coherency.
- Ensure task and system memory protection.

User Memory Space

- Segmentation (per process)
  - Segment base address + segment offset
  - Functionality and type of memory area
  - Assigned accessibility : read-only, executable
  - Segments : text, data, stack, heap
- Pages (per process within each segment)
  - Page base address + page offset
  - Contain actual data (code)

Logical and Physical Memory

- Logical address : 32 bits
  - 20 bits : page base address
    - 1 million pages
  - 12 bits : offset
    - 4KB in each page
- Mapping table
  - 1M entries and 4 bytes in each entry
  - A page base address is the index of the table.
  - Each entry is the actual physical address.

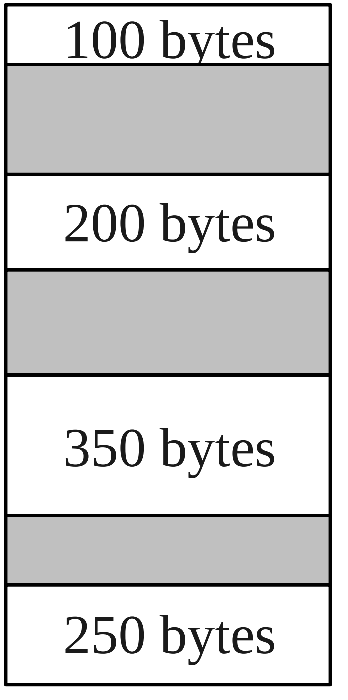
- Example:
  - 1GB physical memory
  - LA: 0xBF8AD73A
  - Base address: BF8AD
  - Offset: 73A
  - Mapping table
    - E[BF8AD]=1D90A000
  - PA: 0x1D90A73A

User Memory Space

- Stack management
- Heap management
  - Algorithms
    - First fit
    - Next fit
    - Best fit
    - Worst fit
    - Quick fit
      - using a more complicated data structure for quick search
  - Fragmentation
    - External : unused after deallocation
    - Internal : unused with over allocation

Heap management

- Example
  - Put 150 bytes into the heap now.
  - Gray area has been allocated.
  - First fit
  - Next fit
  - Best fit
  - Worst fit



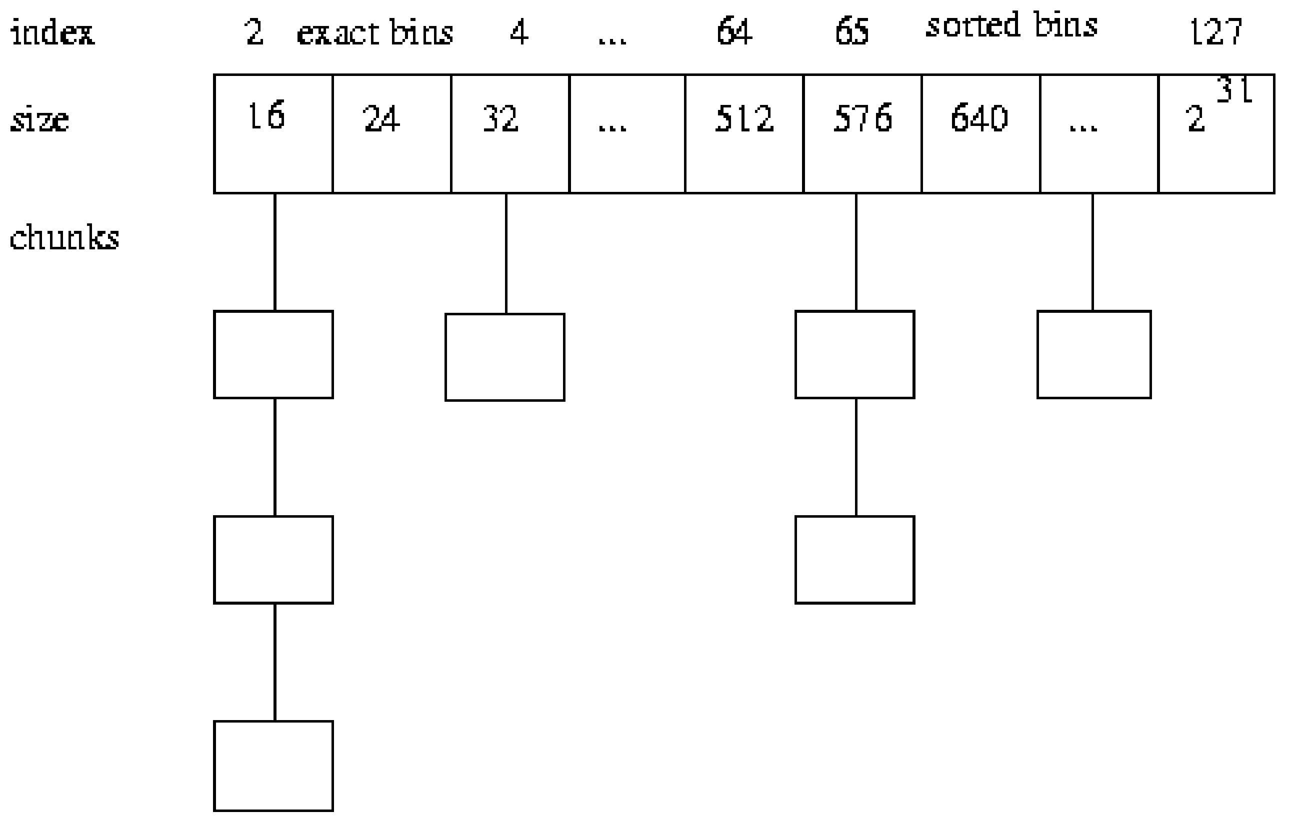
Case 17: Heap in TinyOS

- Data structure
  - A free list : single linked list
  - Free node :
    - size (2 bytes)
    - next pointer (2 bytes)
    - free memory (size-2 bytes)
  - Allocated node :
    - size (2 bytes)
    - allocated memory (size bytes)
- Algorithm
  - Exact match (first round)
  - Worst fit (second round)
  - Allocate and release

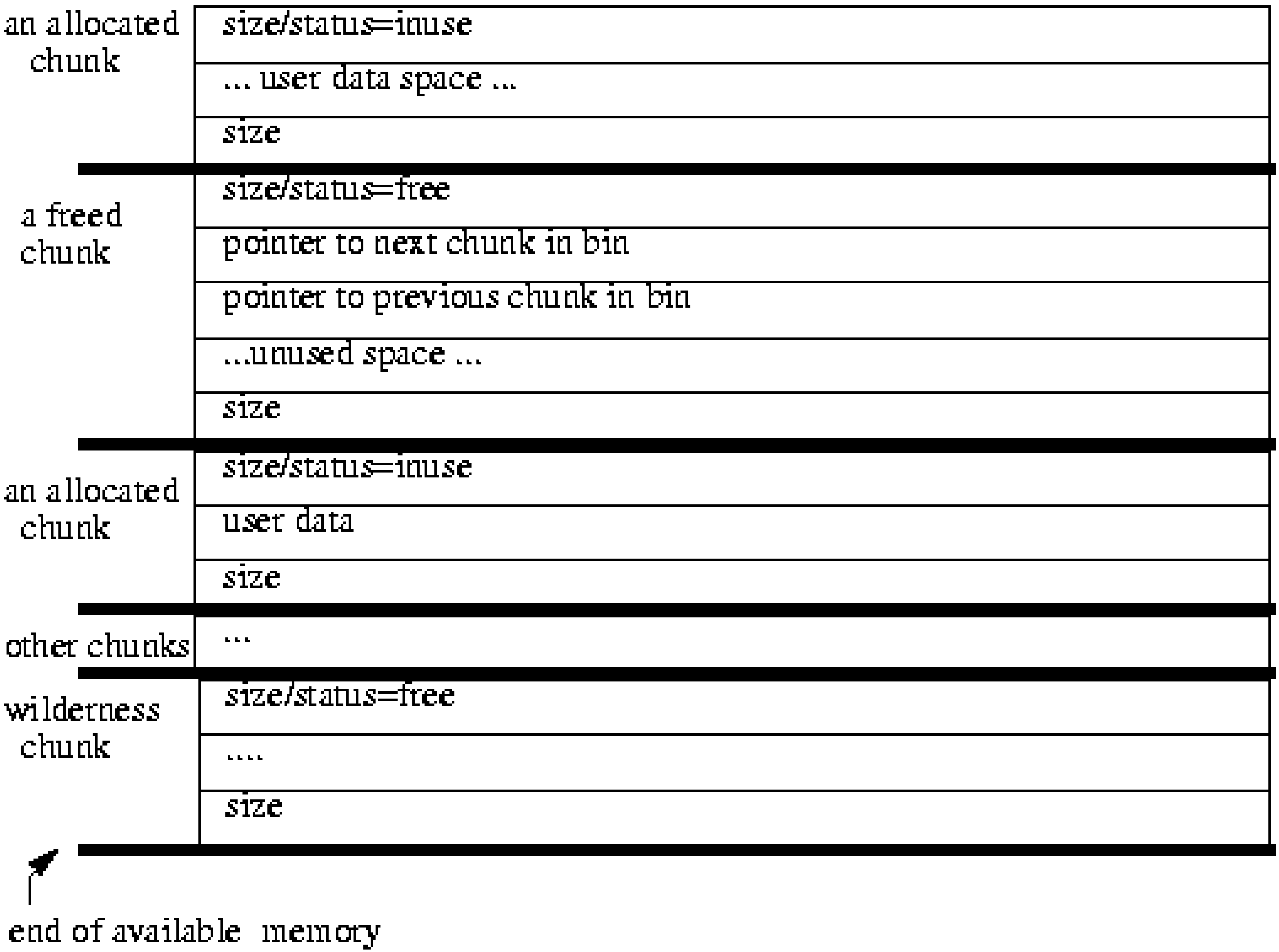
```
struct __freelist
{
  size_t sz;
  point_t nx;
}
```

Heap in Linux

- Memory blocks are called chunks
- Free chunks are grouped in bins.
- Each bin is a double linked list.
- Best-fit



Heap in Linux



Kernel Memory Space

- Kernel code
  - System calls
  - Scheduler
  - Kernel management modules
  - Driver modules
- Kernel data
  - TCBs
  - System call tables
  - Management
    - Sizes of most system data structures are known.
    - Pre-determined stack
    - Buddy system or slab system for dynamic data

Kernel Memory Management

- Buddy system
  - No external fragmentation
  - Easy to implement, for systems without MMU (286)
  - The total memory size is a power of 2.
  - The memory is divided as a binary tree.
  - If n bytes are requested, search in the binary tree such that a free node best fits the request.
  - When a node is released, it will be merged with its buddies upward as much as possible.

Kernel Memory Management

- Buddy system
  - Total 128B
  - Smallest block 16B
  - Request 1: 19B
  - Request 2: 7B
  - Release 1
  - Request 3: 9B

Kernel Memory Management

- Slab system
  - Some kernel data objects that are frequently created and destroyed
    - semaphores, file descriptors, task control blocks
  - Retain an allocated memory that used to contain a data object of certain type
  - Reuse that memory for the next allocations for another object of the same type
  - Others: SLOB and SLUB

Cache

- For systems that have a good locality of references
  - Access most of their data (and code) from a limited section of memory
- Cache stores snapshots of mostly accessed parts of the memory
- Data access
  - Cache hit
  - Cache miss

Cache

- Given (no page replace on miss)
  - The time to access an external address is 10ns.
  - The time to access an internal address is 1ns.
  - The cache miss rate is 1e-3 or 1e-4.
- What is the average access time?
  - 1*0.999+(1+10)*0.001=1.01ns
  - 1*0.9999+(1+10)*0..0001=1.001ns

Cache Page Replace

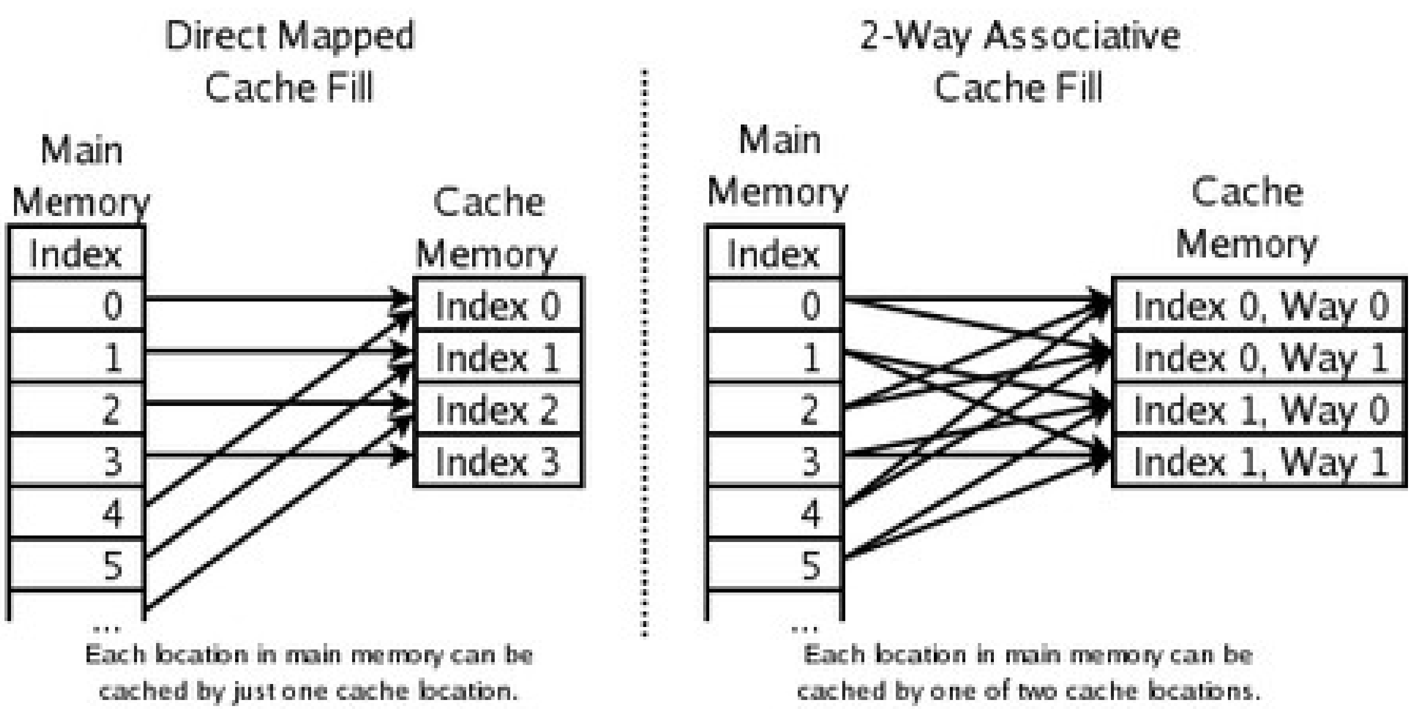
- Given (page replace on miss)
  - The time to access an internal address is 1ns.
  - The time to replace a page is 4us.
  - The cache miss rate is 1e-3 or 1e-4.
- What is the average accessing time?
  - 1*0.999 + (1+4000+1)*0.001 = 5.001ns
  - 1*0.9999 + (1+4000+1)*0.0001 = 1.4001ns

Cache Management

- Address mapping for accessing and replacing
  - 16-bit main memory, 10-bit cache page, 4 pages
  - An address is divided into tag bits, index bits and offset bits.
  - If a page is in cache, its tag bits are stored in the cache as well as the page itself.
  - Search process, given a virtual memory address:
    - Use the index bits to locate the pages in cache.
    - Search if any located page has the tag bits.
    - If yes, the page is in cache and access the page
    - If no, access the page in memory or swap if needed.

Cache Management

- Direct mapped : tag(4) + index(2) + offset(10)
- Set associative : tag(5) + index(1) + offset(10)
- Full associative : tag(6) + offset(10)



Cache Management

- Cache swapping/replacement
  - Least recently used (LRU)
    - Each page has a record of the last time it was accessed.
    - The page of the furthest last time will be replaced.
  - First in first out (FIFO)
    - The indexes of all pages are put in a FIFO queue.
    - The header of the queue will be replaced.
    - The first in does not mean the least recently used.

Cache Management

- Cache swapping/replacement
  - Not recently used (NRU)
    - Each page has a timer that is reset whenever it was accessed.
    - A page with an expired timer will be replaced.
  - Second chance FIFO
    - Each page has a bit set to 1 when being accessed.
    - The bit is set to 0 when being checked.
    - A check is triggered by a cache miss.
    - The page with a 0 bit when being checked will be replaced.

OS Components

- Kernel mode vs. user mode
- Memory management
  - Logic vs. physical memory
  - Memory allocation
  - Cache management
- Process management
  - Multi-process / multi-task scheduling
  - Interrupt and error handling
- I/O system management
  - Shared I/O devices

Task

- Task : process
- A running program in CPU
  - Defined and identified by its context : code and data
  - The kernel maintains a record for each task
- In general computers, a process is normally the execution of an executable file.
- In embedded system, a task is normally the execution of a block of code in memory triggered by an event.

Task

- A CPU can only execute one task at one time.
- Uni-task OS
  - At any time, only one of the task is being executed. A new task will not be processed until the current task is completed.
- Multi-task OS
  - Multiple tasks are processed concurrently.
  - Require complicated task management
    - Task context management
    - Task memory management (only possible with MMU)
    - Task scheduling

Thread

- A task may have multiple threads
  - Threads share inside the task : address space, program code, heap, global data, I/O resources.
  - Each thread has its own : PC, SP, stack, register, scheduling information.
  - Benefits:
    - No special inter-thread communication mechanism is needed.
    - No special shared resource management mechanism is needed.

Multi-task

- The kernel must allocate a certain amount of time in the CPU to execute a task and switch the CPU among tasks.
- Task management
  - Task implementation
  - Task scheduling
  - Task synchronization
  - Inter-task communicaiton

Task Implementation

- Task components
  - In kernel space: TCB (task control block)
    - ID, state, priority, CPU context, allocated memory, ...
  - In user space: code and data
    - Text, BSS, Heap, Stack
- Task hierarchy
  - OS initial task
  - Parent task
  - Child task

Task Creation

- Fork/exec
  - Fork creates a child task as the copy of the parent task.
  - Exec loads a program in the child task to replace the program of the parent task.
  - Changes in kernel space.
  - Changes in user space.
- Spawn
  - Spawn creates a child task as a brand new task.
  - Spawn needs to specify the context of the child task.

Example

- How many tasks for a program foo?
  - What is the graph of tasks in the system?

```
for (i=0;i<3;i++)
{
  fork();
}
```

```
for (i=0;i<3;i++)
{
  fork();
  exec(“foo”);
}
```

Task States

- Ready
  - The task is ready to be executed at any time.
- Running
  - The task is being executed.
- Waiting
  - The task is not ready and waits for some event to make it ready.
- State transition: Figure 9-16
  - Two minor states: new and exit

Task Queues

- Ready queue
  - All ready tasks
- Waiting queue
  - All waiting tasks
  - For each type of event, the waiting queue is divided into sub queues.
    - A waiting queue for key board
    - A waiting queue for flash drive
- Queue management

Examples

- TinyOS task scheduler
  - Only has a ready queue, no other queues
  - tos.system.SchedulerBasicP
    - Task queue
      - uint8_t m_next[NUM_TASKS];
    - Task dispatch
      - Scheduler.taskLoop()
  - What does an interrupt do to task management?
    - What happens when a packet is received?
    - What happens when two packets are received?
  - “In TinyOS 2.x, a basic post will only fail if and only if the task has already been posted and has not started execution.” (TEP106)

Task Scheduling

- Scheduler
  - An endless loop dispatches ready tasks to the CPU according to some scheduling algorithms.
- Scheduling criteria (per task)
  - Response time
    - From the time the task is created to the time the task is being executed.
    - States: time from new to running

Task Scheduling

- Scheduling criteria (per task)
  - Turnaround time
    - From the time the task is created to the time the task is completed.
    - States: time from new to exit
  - Waiting time
    - The sum of time of a task spent waiting in the ready queue.
    - States: time in ready (not waiting)

Task Scheduling

- Scheduling criteria (per system)
  - Overhead
    - The time and data needed to determine which task will run next.
    - It is a cost on scheduler, not on a particular task.
  - Throughput
    - The number of tasks completed per unit time.
  - Fairness
    - The time percentage of running allocated to a task.
    - Starvation : a task never gets a splice of running time.

Task Scheduling Algorithms

- Given a sequence of tasks
  - The time a task is created
  - The duration a task needs
  - The priority a task has
- Given a configuration of a scheduler
  - The time slice for each task
  - The scheduling algorithm
- Show how the tasks are executed!!!

Task Scheduling Algorithms

- Assume three tasks T1, T2 and T3
  - T1 needs 20s
  - T2 needs 10s
  - T3 needs 7s
  - At t=0, T1 comes to the system
  - At t=7, T2 comes to the system
  - At t=14, T3 comes to the system

Non-preemptive Scheduling

- Non-preemption
  - A tasks is **not forced** to give up the control of the CPU before it is finished.
- FCFS (first come first serve)
  - Only ready queue, no waiting queue
  - State transition
- SPN (shortest process next)
- How tasks are executed and what are the performance of the two algorithms?

Preemptive Scheduling

- Preemption
  - A tasks is **forced** to give up the control of the CPU before it is finished.
- Round Robin/FIFO
  - Each task in the ready queue is allocated an equal time slice.
  - A running task is preempted at the end of its time slice by a timer interrupt.
  - The preempted task is added to the end of the ready queue.
  - The algorithm has a constant overhead for context switch.

Preemptive Scheduling

- Priority Scheduling
  - Basic algorithm
    - All tasks are assigned priorities.
    - A task with a higher priority will be executed first when ready.
    - Tasks with lower priority will be preempted.
  - Starvation
    - A task with lower priority will age as its time in the ready queue grows.

Real Time Operating System

- General idea: a task has a deadline to meet.
- A task missing the deadline is as bad as the task is wrong or not finished.
- Examples
  - Periodic sampling : the processing of the current sample must finish before the start of the next sampling.
  - ABS : the control signal must be sent out to the brake within 0.1s to release the brake after the pedal is pressed. (If a car is at 60mph, 0.1s means about 9 feet.)
  - Chess game : the player (computer) must decide the next step before time is out.

RTOS vs. General OS

- RTOS is deterministic in that the worst case response time to an event is known.
- When an event happens, an interrupt is raised.
  - In GenOS, the interrupt maybe deterred for an unknown period by an ongoing uninterruptable execution of code.
  - Example
    - A normal process is waiting for reading a file.
    - An emergency happens that must read a patient information and the normal process cannot block the emergency process.

RTOS

- When an event happens, an interrupt is raised.
  - In GenOS, the latency of interrupt handling and task context switching may exceed the deadline requirement of a task.
  - Example
    - When an interrupt happens, is it necessary to save all information of a running process?
    - Optimize the information of a running process.
    - Optimize the interrupt routine to use least resource that may conflict with a running process.
    - Make each interrupt routine deterministic.

RTOS Characteristics

- When an event happens, an interrupt is raised and a task is scheduled for processing.
  - The task with higher priority can preempt other tasks at any time.
  - The worst case execution time of the task is known.
  - The interrupt latency is minimum and known.
  - The task context switching time is minimum and known.
- Although it is called RTOS, it is not guaranteed a task will be completed in real-time (before deadline).
  - A system is dealing with multiple tasks at the same time.
  - Tasks can be preempted by other tasks.

RTOS Scheduling

- Earliest deadline first (EDF)
  - OS knows the deadline of all tasks.
  - The task with the closest deadline has the highest priority.
  - Figure 9-25
- EDF with dynamic voltage scaling
  - Assume all tasks are known with worst execution time C_i and period P_i.
  - Tasks are schedulable without DVS if $\sum \frac{C_i}{P_i} < 1$
  - Tasks are schedulable with DVS if $\sum \frac{C_i}{P_i} < \frac{f_s}{f_M}$

RTOS Examples

- vxWorks
  - Preemptive priority and round robin
  - Applications (wiki): BMW iDrive system, Linksys WRT54G wireless routers, Apache Longbow attack helicopter, Deep Impact space probe, Phoenix Mars Lander
  - Bought by Intel for its embeded system product line in 2009.
- Jbet
  - EDF
- Linux
  - Similar to VxWorks
  - The book is out of date

Application Examples

- BMW iDrive (wiki)
  - The first generation iDrive system is based on Microsoft's Windows CE for Automotive. This can easily be seen when the system reboots or restarts after a software crash displaying a "Windows CE" logo.
  - However, starting with the second generation (first seen in the 2004 5-series), Microsoft Windows CE was replaced with Wind River's VxWorks, a real-time operating system.
  - The third generation will employ Internet Protocol networking, using off-the-shelf IP components to replace proprietary networking systems.

Inter-task Communication

- ITC, also called IPC in general OS
- Mechanisms
  - Without MMU
    - Global data area
  - With MMU
    - Memory sharing
    - Message passing
    - Signaling

ITC

- Memory sharing (with virtual memory spaces)
  - All tasks are given pointers pointing to the same memory area.
  - Race condition :
    - Task A writes data and then is preempted by task B.
    - Task B writes data and then finishes.
    - Task A reads the data.
  - Mutex/semaphore : at any time, if a task is accessing the memory, it locks the memory such that other tasks cannot access the memory.
    - The task requests a mutex/semaphore from the system that provides hardware-support to prevent race condition on mutex/semaphore.

ITC

- Semaphore
  - An integer s: if s==0, the share memory is locked.
  - Opertions to the integer s is atomic (neither interruptable nor preemptable)
  - General operations
    - Init (sem_init()): s=1
    - Wait and acquire (sem_wait()): wait if s==0, then s--
    - Release (sem_post()): s++
  - Only the task that is owning s can access the share memory.

ITC

- Dead lock among tasks
  - Example
    - Task A writes data and then is preempted by task B that has a higher priority.
    - The data area is locked by A.
    - Task B needs to write data to the memory, but has to wait for A to release.
    - Task A has to wait for B to complete, because A's priority is lower than B.
  - Various mechanisms developed in OS

ITC

- Message passing
  - Two processes work as a client and a server.
  - OS specifies the addressing and communication protocols.
- Signalling
  - A process registers signals and signal handlers.
  - Upon receiving a signal from another process, the corresponding signal handler will be invoked by the OS to process the signal.
  - Signals are not necessarily interrupts.

OS Components

- Kernel mode vs. user mode
- Memory management
  - Logic vs. physical memory
  - Memory allocation
  - Cache management
- Process management
  - Multi-process / multi-task scheduling
  - Interrupt and error handling
- I/O system management
  - Shared I/O devices

I/O and File Management

- Storages : flash or hard drive
- Memory mapped storage
  - Map file systems onto memory.
  - Support primitives for manipulating files and directories.
    - File attributes : name, type, size, access permission, ...
    - File operations : create, open, read, write, close, ...
    - File access methods : sequential and random

OS Standards

- Standards for the interface between OS and applications (API)
- POSIX (Portable Operating System Interface)
  - Specify the APIs that an OS should provide to any application.
    - Functions : arguments, return, errors, exceptions
    - Functionality of functions
  - Do not specify how a system function is implemented.

POSIX

- Functions
  - Thread management
  - Semaphore management
  - Priority management
  - Scheduling management
  - Process management
  - Memory management
  - I/O management

POSIX

- Published by IEEE
- http://www.opengroup.org/onlinepubs/009695399/mindex.html
- Base definition
- System service functions
- Other rationales

Final Notes

- Application development
  - No major difference to application development in general computer system
    - Software life cycle : design, implement, test, ...
  - Factors to be considered
    - Resource : memory, CPU time
      - Don't simply declare local variables or dynamic arrays.
      - Don't assume something should happen by now.
    - Design of OS and drivers
    - Cost : $

Final Notes

```
foo() {
  int a[200];
}
```

```
foo() {
  call Packet.send(p1);
  call Packet.send(p2);
}
```

- Problem in sensor?
- Problem in TinyOS?