# Topic 1: Basic Concept

## Overview

### Guidance

* Chapter: Operating-System-Concepts.pdf 1, 2.2, 2.3; Unix-Network-Programming-Vol.2 Part 1
* Spend time: 2 days
* Learn suggestion: Take a general understanding of the operating system
* Key points: POSIX, System Call, Signal, Timer, File Descriptor
  + What's the OS? Why do we need OS?
  + The history of Unix, What's the relationship between Linux and Unix?
  + What's the fundamental function of OS?
  + Try to get help from the Linux help manual
  + Write some programs for each type of system call using the function in page 68.

### Practice

None

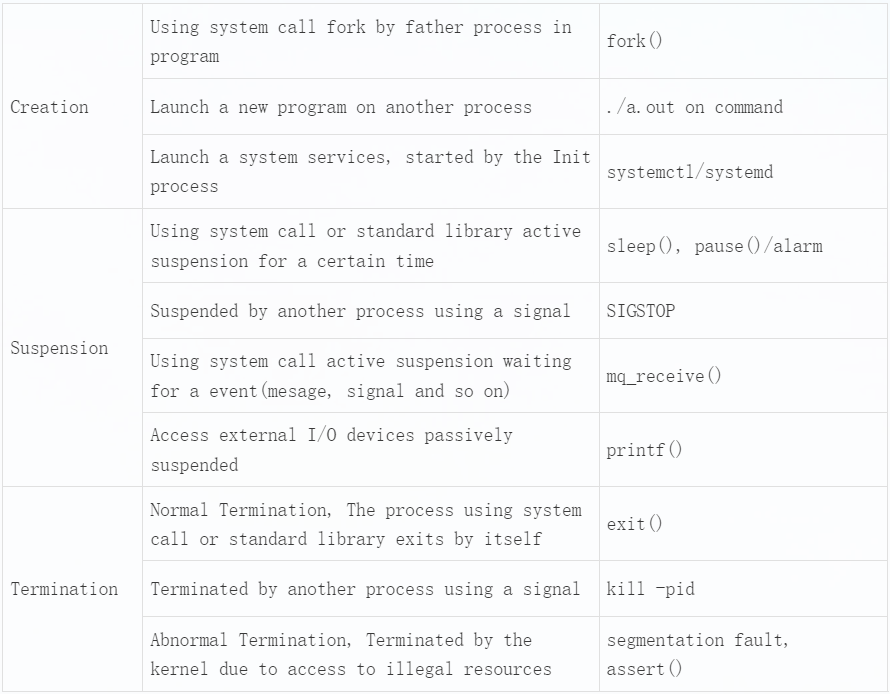
## Process

### Guidance

* Chapter: Operating-Systems-Internals-and-Design-Principles.pdf 3
* Spend time: 1 day
* Learn suggestion: To know about PCB, Process state, Process creation/termination
* Key points:
  + Using command ps/top to check the information of process, Try to explore what options can be used
  + How to capture the different process state?
  + The memory layout of Process address space

### Practice

1. How many approaches can a process be created, suspended, or terminated? Illustration



1. What's the meaning of the zombie and orphan process? Create a zombie and orphan process separately and check it by command ps

Orphan process: When a parent process exits while one or more child processes are still running, they become orphan processes. Orphan processes are adopted by the init process (process id 1), and the init process collects status on them.

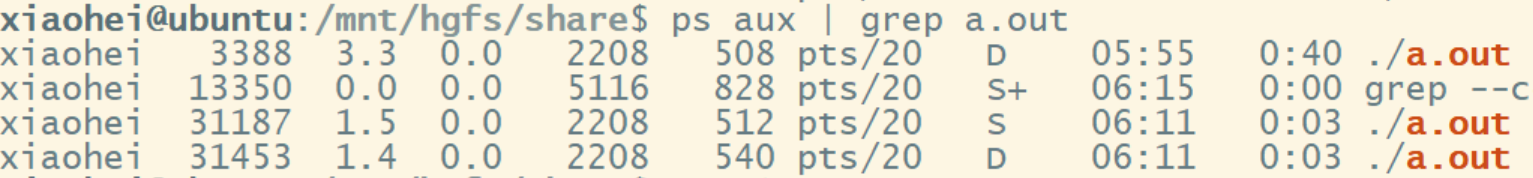
Zombie process: A process creates a child process using fork. If the child process exits and the parent process does not call wait or waitpid to get the state information of the child process, the process descriptor of the child process is still saved in the system. This process is called a dead process.



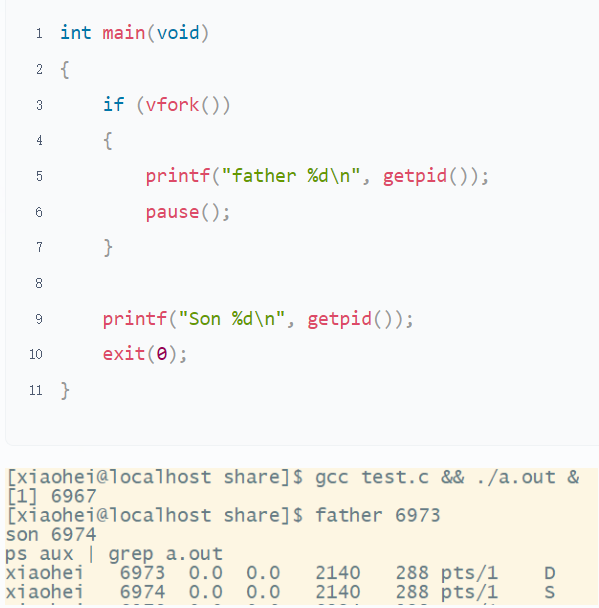


1. Try to get the state TASK\_UNINTERRUPTIBLE(D) of process in linux
   1. Prolongs the waiting time of a process in the task queue





* 1. vfork(Not uninterruptible)



* 1. Using kmod to lock kernel processes.(Leave it for you to explore)

## Thread

### Guidance

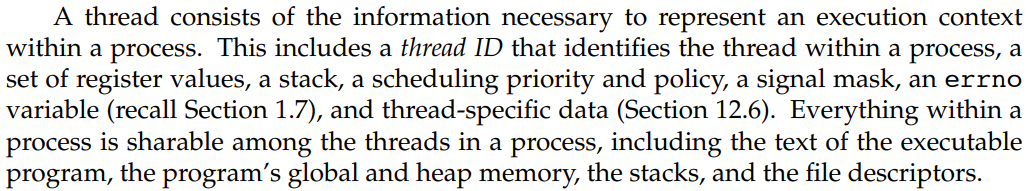
* Chapter: Operating-System-Concepts.pdf 4
* Spend time: 1 day
* Learn suggestion: Learn to compare with process and coroutine
* Key points:
  + Difference between process and thread
  + Difference between ULT and KLT
  + How many ways to implement the ULT?

### Practice

1. For multiple threads in the same process, which resources are shared and which are unique?

* Resources exclusive by thread

Advanced-Programming-in-the-UNIX-Environment P284



note: The global variable can become thread exclusive after declared by \_\_thread.

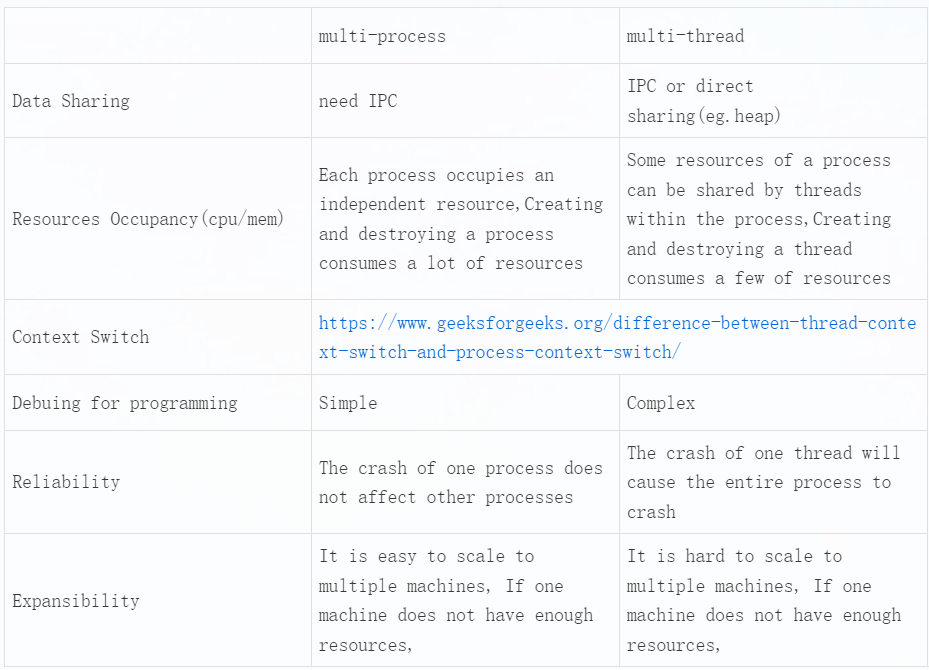
* Resources shared by threads

Global variable table(.text, .data, and so on), heap, dynamic-link library, files, Current working directory, as well as user ID and group ID

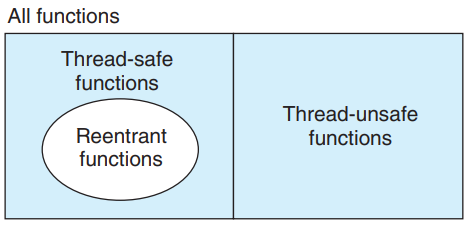
1. What are the advantages and disadvantages of multi-process(only one thread in each process) and multi-thread programming?

Process: Minimum unit of resource allocation

Thread: Minimum unit of cpu scheduling



1. What are reentrant and non-reentrant functions? What are thread-safe and thread-unsafe functions? Illustration



Reentrant functions and Thread-safe functions are different concepts. Refer to p1059 on "Computer-Systems-A-Programmers-Perspective.pdf"

Reentrant functions: Refer to chapter 10.6/12.5 on "Advanced-Programming-in-the-UNIX-Environment.pdf"

Thread-safe functions: When a function is called by different threads at the same time, each thread will get the correct result. This can only occur in multi-threaded scenarios.

Reentrant functions: If a function is interrupted at any time during execution, it can still be reused, or the correct result can be obtained after the interrupt is returned. This can occur in multi-threaded or single-threaded scenarios.

Determine non-reentrant functions

* Functions that directly use or return global/static variables, including the errno variable
* malloc/free is called
* Standard I/O function is called， most of which use global data structures in a non-reentrant manner
* Other non-reentrant functions

A non-reentrant function can be locked to become thread-safe, but it's not reentrant. Reentrant functions are higher-level thread-safe functions, a concept similar to asynchronous signal/interrupt safety functions.

The function is a reentrant if its data is in its own stack space.

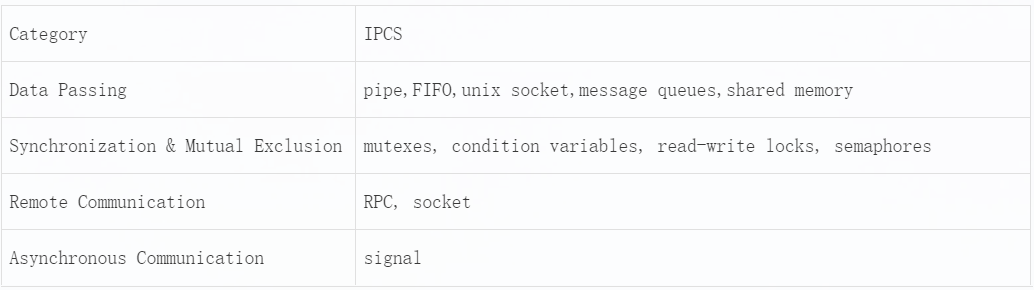
## IPC

### Guidance

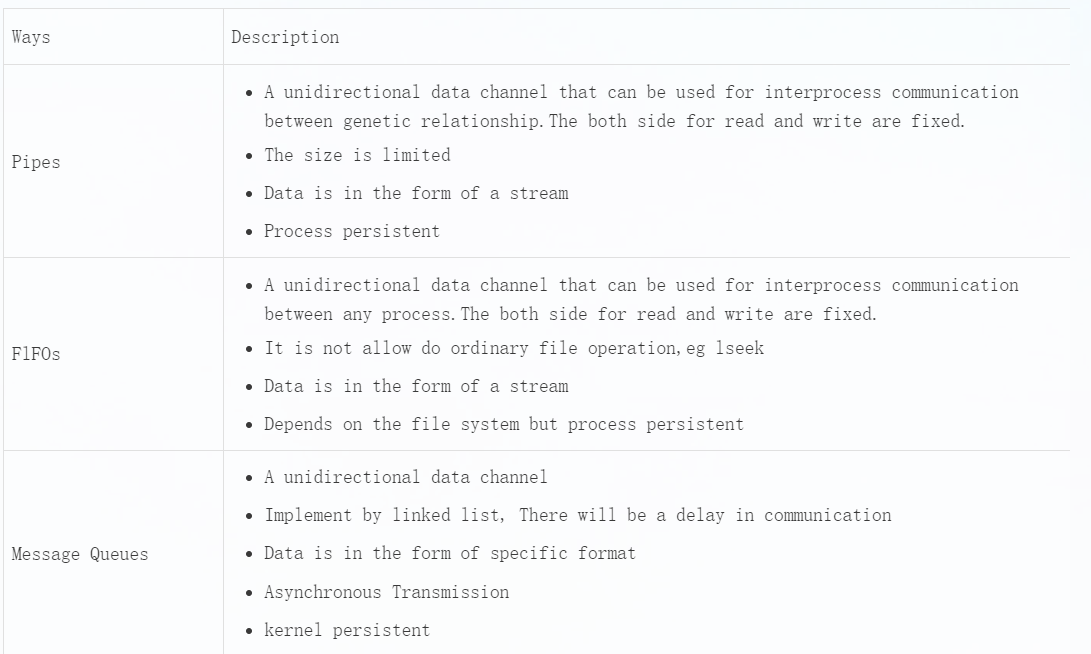
* Chapter: Unix-Network-Programming-Vol.2.pdf Part 2-5; Advanced-Programming-in-the-UNIX-Environment chapter 17
* Spend time: 2 days
* Learn suggestion: Compare and understand the characteristics of communication modes between different processes
* Key points:
  + How many ways to communicate between different processes?
  + The life cycle of the IPC object
  + What is different between Posix and System V IPC?
  + Can these ways be used for inter-thread communication?
  + What are the characteristics of each IPC?

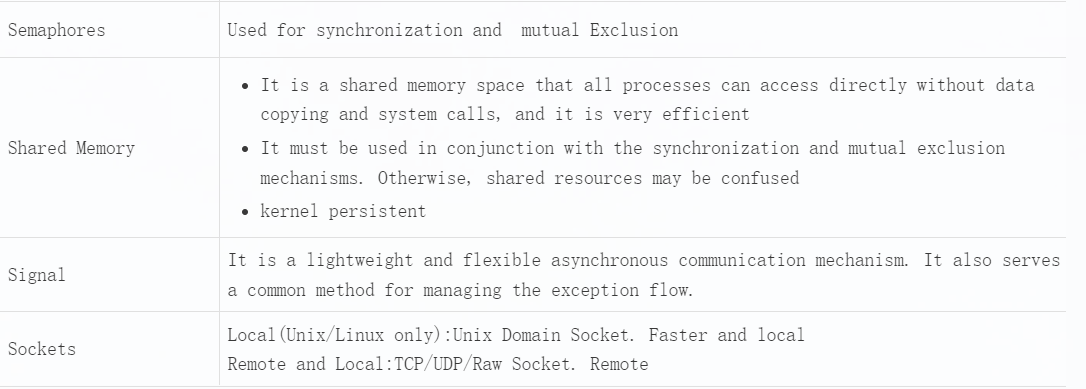
### Practice

1. According to your understanding, try to divide all IPCs into different categories, and the reason



1. To summarise, what are the characteristics of each IPC?





1. Using fork/exec to simulate executive "ping 127.0.0.1 -c 3" and get the result

Refer to "fork-exec simulate executive ping.c"

1. Write a program to implement the synchronisation and mutual exclusion by semaphore or lock

Refer to "Synchronisation and Mutual Exclusion.c"

1. Follow the client/server model: client sends a filename to the server, then the server responds its content to the client(Unix Domain Socket)

Requirements:

* Return the state of server to client when the specified file can't be opened correctly
* Follow the TLV format to organise data
* Don't mix the file operation and socket operation together. Separate them

Refer to "UDP based domain socket"

## Signal

### Guidance

* Chapter: Advanced-Programming-in-the-UNIX-Environment.pdf 10.1-10.10
* Spend time: 1.5 days
* Learn suggestion: Have a general understanding of the signal and where it's being used. Don't need to learn the advanced usage of it for now.
* Key points:
  + How many signals can be used? What's the meaning of each signal?
  + How to check all signals in Linux? Where is the signal 32/33?
  + What's the difference between signal and interrupt?
  + What's a real-time signal and a non-real-time signal?
  + How do you control the creation of core dump files when a program crashes?

### Practice

1. Modify the default behavior of ctrl + c when ctrl + c is pressed, the output is Hello World! ; To achieve the same effect by the command kill



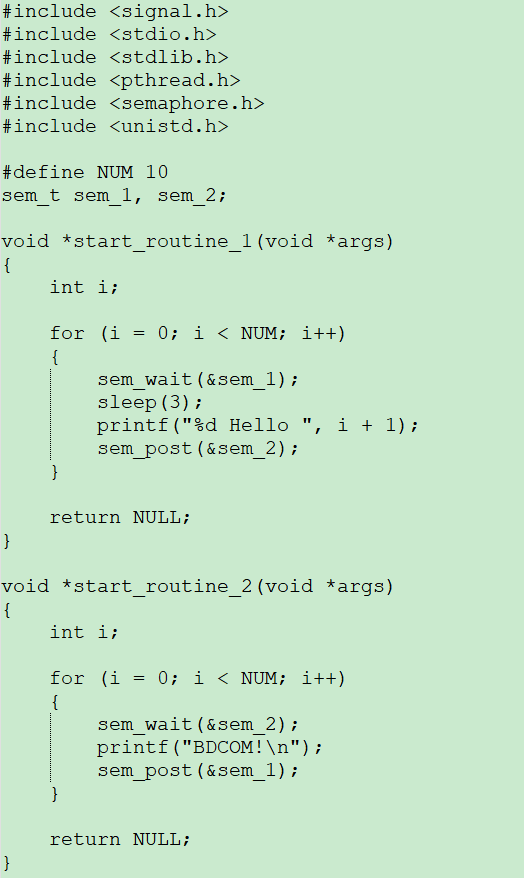
1. Write a program to simulate the try/catch/finally in C. It's able to handle both file inexistence and segment fault

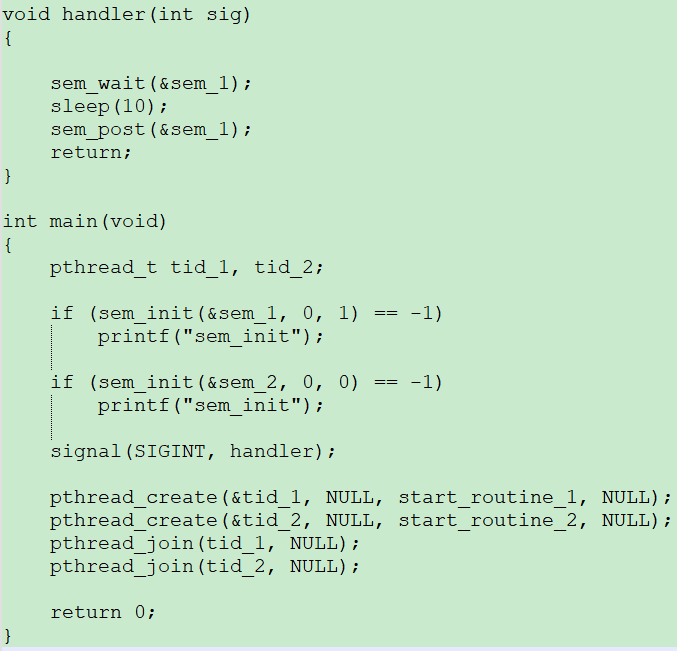
Refer to "Implement to simulate the try-catch-finally.c"

1. Implement the function sleep() and a periodic timer using alarm() and pause() simulations. And wake up the process from the sleep state in advance in the timer periodically

Refer to "Implement to simulate sleep and alarm by signal.c"

1. What issues does signal may bring to the program? Illustration
   1. Disturbs the program execution sequence
   2. Disrupts program logic, causing resources to wait for each other





* 1. A non-reentrant function is called in both a normal function and a signal handler at the same time, which may lead to a data mess

# Topic 2: VxWorks Task

## Task

### Guidance

* Chapter: Vxworks-programmers-guide-5.5.pdf 2.2 & MIPS.CVM/docs/libIndex.htm
* Spend time: 1.5 days
* Learn suggestion: Learning it to compare with the Process and Thread
* Key points:
  + What's the difference between spawning a task at the system boot and the command line?
  + What happens if the task does not exist when deleting a task by task ID?
  + What's the meaning of the fields displayed in the command "show task"?

Experiment

1. Create and run a task when booting the switch
   1. Create and run a task named DM1 with stack size 1024 bytes and priority 128 in function os\_demo\_init. Print the task ID and name here
   2. DM1's start routine is os\_demo\_main\_process. The function has a main loop, which will be active for 10 seconds.
   3. There is a global variable named g\_int\_data, which will increase by 1 and print the task ID and name every time the main loop is activated.
2. Create and run a task when executing a command
   1. Create and run a task named DM2 with stack size 1024 bytes and priority 128 in function os\_demo\_cmd\_impl\_test1 of file "os-demo-cmd-impl.c", when the command "os-demo test1" is executed. Print the task ID and name here
   2. DM2's start routine is os\_demo\_cmd\_impl\_test1\_process. The function has a loop. It will be active for 30 seconds.
   3. Print the task ID, name, and the global variable g\_int\_data every time the loop is activated.
3. Destroy a specific task
   1. Reconstruction function second\_os\_demo\_config\_test2 to accept a number
   2. Delete a specified task ID, which is prefixed with DM

### Practice

1. Is it possible for DM1 and DM2 to access both the global variable g\_int\_data? Why?
2. What is the meaning of in following fields that display on "show task"?

NAME ENTRY TID PRI CPU invoked

1. What happens when the command "os-demo test1" is executed multiple times?
2. When deleting a task by task ID, what happens if the task does not exist? How to prevent this phenomenon from happening. Hint: Find the pattern, which kind of addresses are causing problems, and why.

Refer to "Task creation and deletion"

## Inter-Task Communication

### Guidance

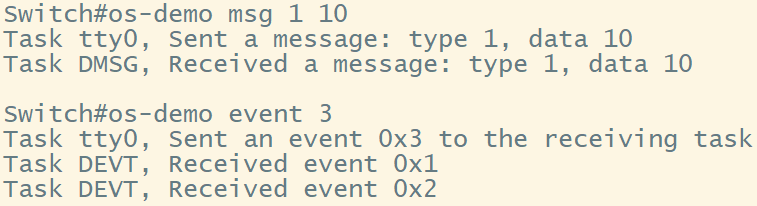
* Chapter: Vxworks\_programmers\_guide-5.5.pdf 2.3, 2.4
* Spend time: 1 day for each of the experiments
* Learn suggestion:
  + Don't waste too much time on the switch command line. Get the code for me if you can't implement it after you spent much time
  + Allocate the stack size properly
  + Implement synchronization and mutual exclusion by semaphores and data passing by message queue or event

Experiment

If not specified, the default values for the tasks are as follows： stack size is 1024 bytes, priority 128, no parameter

### Experiment with messages and events in the different tasks

1. Message and Event receiving
   1. Create a message queue whose length is 20, as MSG\_Q\_FIFO mode
   2. Create and run a task named DMSG, which starts the routine os\_demo\_msg\_recv and is used to receive messages.
   3. Create and run a task named DEVT, which starts the routine os\_demo\_event\_recv and is used to receive events.
2. Message and Event sending
   1. Send a message by command line "os-demo msg "
   2. Send an event by command line "os-demo event "
3. Process demonstration



Refer to "Receiving message and event in the different tasks"

### Experiment with message and event in the same task

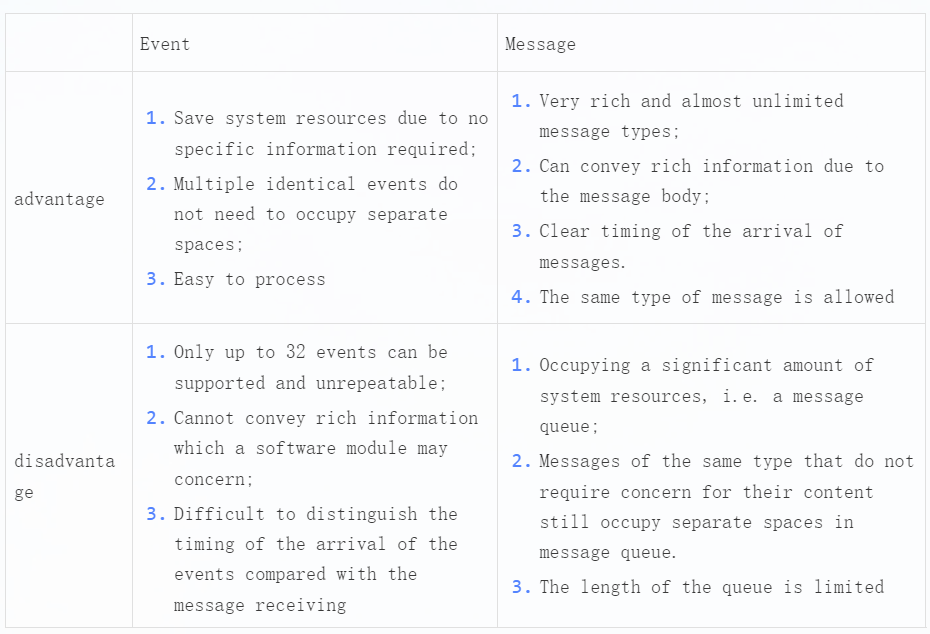
1. Message and Event receiving
   1. Create a message queue with a length is 20, in MSG\_Q\_FIFO mode
   2. Create and run a task named DMNE. The task's start routine is os\_demo\_msg\_event\_recv, and it is used for receiving messages and events.
2. Message and Event sending
   1. Send a message by command line "os-demo msg "
   2. Send an event by command line "os-demo event "

Requirements:

* The task can only have one block point
* The processing of messages and events needs to be as timely as possible

Observation and Thinking

How many solutions can you think of? What are the advantages, disadvantages, and differences between event and message?



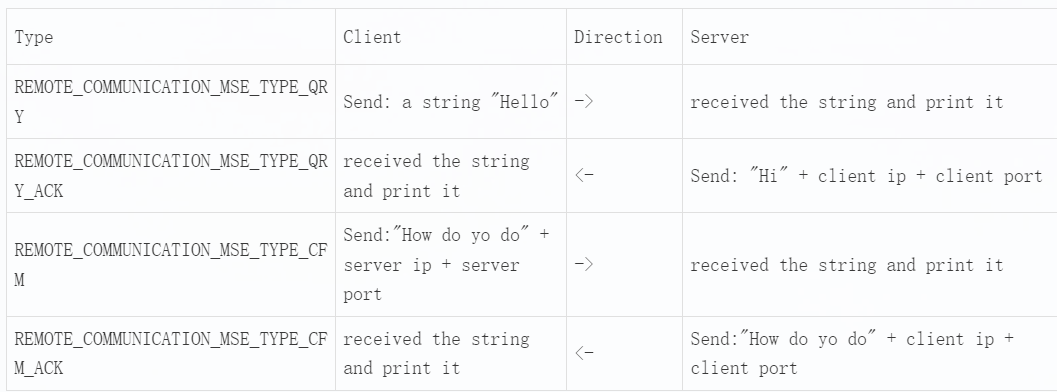
Refer to "Receiving message and event in the same task"

### Experiment with multiple sockets in different tasks

Refer to "socket.c" and "file-write-read for VX switch.c" under the sample folder

1. Implement a 1-to-1 client/server program based on the following requirements. The Python code has already been provided by the client, which runs on Windows

Implement the server program code in C that runs in VxWorks. Create and run a task named SCKT, whose start routine is os\_demo\_socket\_text\_process,stack size is 1024 \* 3 bytes

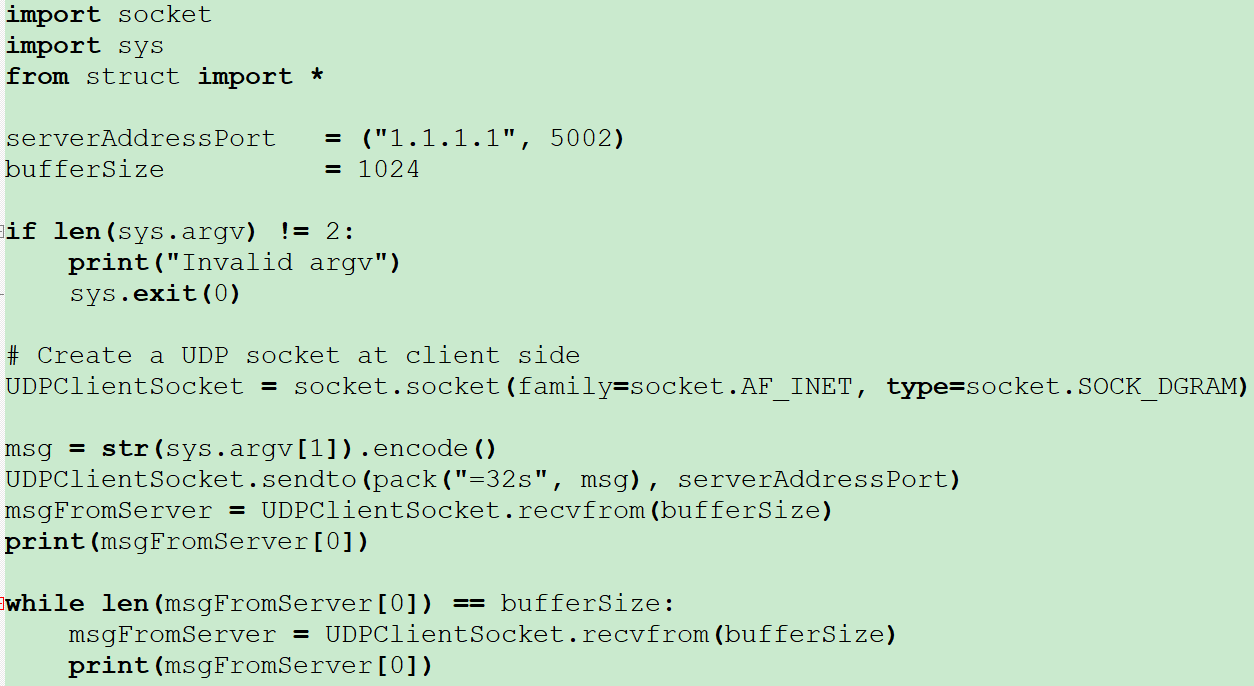


The client implemented in Python



1. Remold question 5 of IPC to make it work on VxWorks.

The client implemented in Python



Refer to "Receiving multiple sockets in different task"

### Experiment with multiple sockets in the same task

Based on system call select() to reconstruct "Experiment with multiple sockets in different task".

There is a task which name is SKTF and start routine is os\_demo\_socket\_text\_file, stack size is 3 \* 1024 bytes. It's waiting for response from file request and text request

Refer to "Receiving multiple sockets in the same task"

### Experiment with message, event and multiple sockets in the same task

Based on socket\_register() to reconstruct "Experiment with multiple sockets in the same task" and "Experiment with message and event in the same task"

There is a task named SKEM, and the start routine is os\_demo\_socket\_message\_event\_process; the stack size is 1024 \* 8. It's waiting for a response from a message, an event and multiple sockets

Refer to "Receiving message, event and multiple sockets in the same task"

## Timer

### Guidance

Refer to the head file and bdip-task.c

### Experiment with the timer using

Reconstruct the experiment "Experiment with message and event in the same task" using the timer instead of the command line to periodically send the event and message to the task, the DMNE.

1. Create and start a timer named timer\_event, which is a loop and callback type. Sent an event to the task DMNE every 15 seconds
2. Create and start a timer named timer\_msg, which is a loop and message type. Sent a message to the task DMNE every 15 seconds
3. Print the information of the receiver.

eg. Task, Received a message or event: type, data

Observation and Thinking

1. What is the difference between the 2 timers?

The message method can only send a message, while the callback function can execute other operations besides sending a message. Additionally, the information contained in the message in the message method is fixed, whereas it varies in the callback method.

1. Can you infer how the timer works?

\_NTM

From the task name printed in the callback function, i.e., “\_NTM”, we can infer that it is a specialized timer task that manages all the timers, including managing the elapsed time of each timer. The timer task determines whether a timer expires during its countdown. When a timer expires, the timer task calls the callback function or sends a message to a message queue, which is specified by the user task (or module) creating this timer. When a user task (or module) calls sys\_add\_timer(), a global timer control block is allocated in the timer pool, and the timer task then manages it.

1. What do you need to pay attention to when you do something on a timer? (Resource management perspective)
   1. As little as occupying the callback function.
   2. Don't operate your structure in the callback function.
   3. Add first, then start.
   4. A loop flag should be added when you need a loop timer.

### Experiment with the timer application

1. There is a linked list that stores many IP cache nodes. Refer to Sample 1 for the structure
2. There is a task named DMCR; the start routine is os\_demo\_ip\_cache\_create\_process, used to create an IP cache node and add it to the linked list. Every random time(refer to sample 2). An IP address is also generated randomly (refer to sample 3).
3. There is a task named DMMC. The start routine is os\_demo\_ip\_cache\_get\_mac\_process, which provides a Mac address for the appointed IP address when DMCR creates the IP cache node. The rule of MAC is that the first 4 bytes are the same as the IP obtained from DMCR, and the following 2 bytes are random.
4. There is a task named DMAG. The Start routine is os\_demo\_ip\_cache\_age\_check\_process, which periodically (implemented by a timer) increases the field age by 1 for each IP cache node and deletes it when the age reaches 100.



Refer to "Experiment with timer application"

## Task Scheduling

### Experiment for the preemptive priority scheduling

Get the corresponding code from "Experiment with the preemptive priority scheduling" and run it

There are 3 tasks:

* Counting task with priority 128. It has two parts of code in its main loop: one is a non-blocking code block implemented as a dual loop, in which the value of a global variable “test\_count” is incremented; and the other is just sleeping by calling taskDelay().
* A checking task with the same priority of 128. It periodically checks the value of test\_count and prints it out.
* A checking task with a higher priority of 126. It also periodically checks the value of test\_count and prints it out. Read the source code for the details.

**Observation and Thinking**

What did you observe? Try to explain

Task DNCK always prints out 640000, while task DHCK prints out a value that varies from 0 to 640000(the probability of printing out 0 or 640000 is very low for task DHCK).

Under the preemptive priority scheduling, a task with a higher priority entering the ready state always preempts the CPU from the current task running. However, a task with the same priority cannot preempt the CPU from the running one.

### Experiment with CPU Hog

Get the corresponding code from "Experiment with CPU Hog" and run it

If a task is overloaded after receiving a message. Although there may not be such a simple dual loop logic in the real system, the situation is similar when dealing with large capacity tables.

Observation and Thinking

1. What did you observe? Try to explain

Unable to interact with the switch through console and telnet,because of the CPU hog caused by the task DMRV, the other tasks have no chance to run.

1. How to optimize this code to eliminate CPU hog

To yield the CPU for a moment every few seconds using tickGet() .

At least the absolute CPU hog is eliminated, and we can access the switch by console and telnet session, although the interaction in the sessions is delayed for response. Because the task DMRV will be in the delay state (i.e., yields the CPU) for a while every time it runs for 1.5 seconds, the relevant tasks now have a chance to run.

Refer to "Experiment for CPU Hog/os\_demo-optimized .c"

# Topic 3: Memory Management

## MBLK

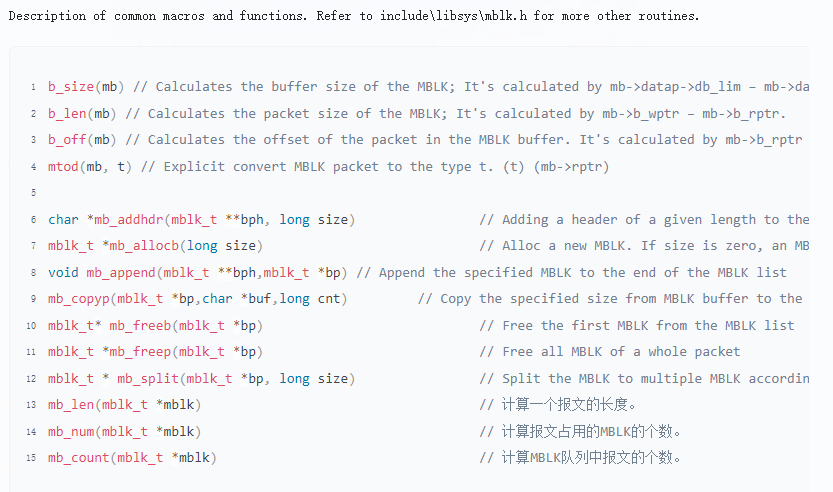
### Description

MBLK, which stands for Message Block, is a data format for network devices to send and receive data.

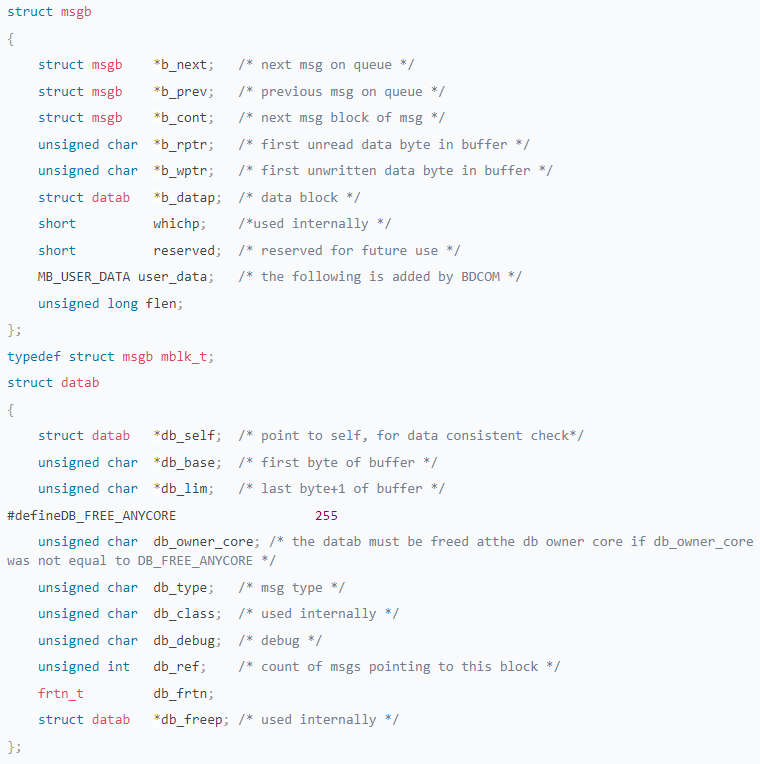
Note that the pointer b\_cont links different segments of a packet, while the pointers b\_next and b\_prev link different packets. MBLK is used to achieve zero-copy packet data and improve packet processing efficiency.

This is mainly reflected in two aspects:

1. When a packet passes through the network stack, it does not need to be copied layer by layer; it is accessed by moving the read or write pointer. When a new protocol header needs to be prepended to the packet, a new MBLK structure can be added to link to the current packet MBLK chain using the pointer b\_cont.
2. As a logical packet is represented by an MBLK, a single packet can be duplicated into multiple packets with the same data for multiple modules to use simultaneously just by adding some objects of type mblk\_t without copying the packet's true data part. These mblk\_t objects point to the same dblk\_t object, and the value of field db\_ref reflects the current reference count of the data buffer of the packet, while each object can have its own read and write pointer values.



### Data Structure



### Packet Parsing

* Guidance

RFC Document: Ethernet/IP(v4), DIX frame/rfc791

* Steps

Replace the library libip.a I provided you in path 220D-L2-REV2\_123631\apps\2228P\lib. Don't delete the old libip.a, Just rename it.

Add a function void os\_demo\_mblk\_handle(mblk\_t \*mb) in file os-demo.c. And print something in this function.

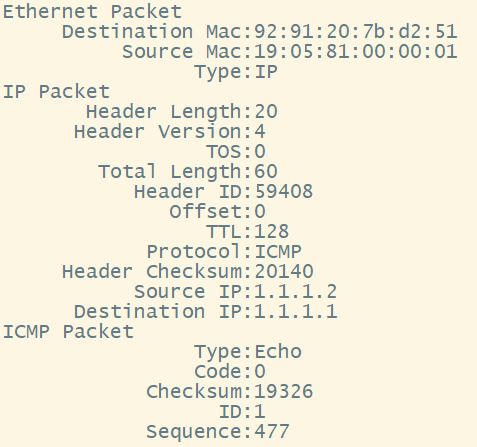
Then, ping the switch.

* Requirements

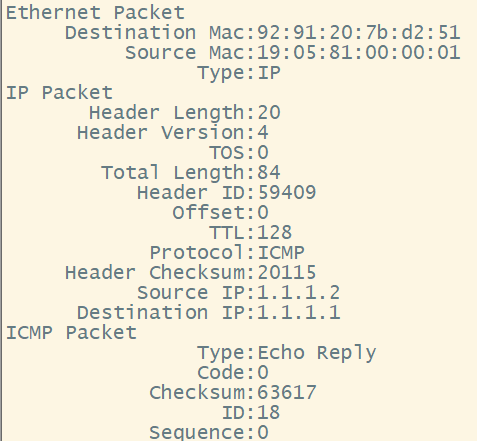
Parse mb to get all the data in it and print out. Compare with the packet captured from Wireshark

* Output

Echo



Echo Reply



Refer to “Parsing the MBLK.c”

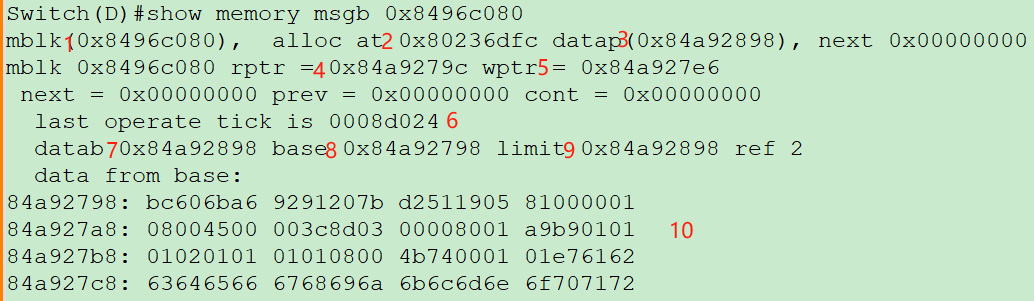
### MBLK Analysis

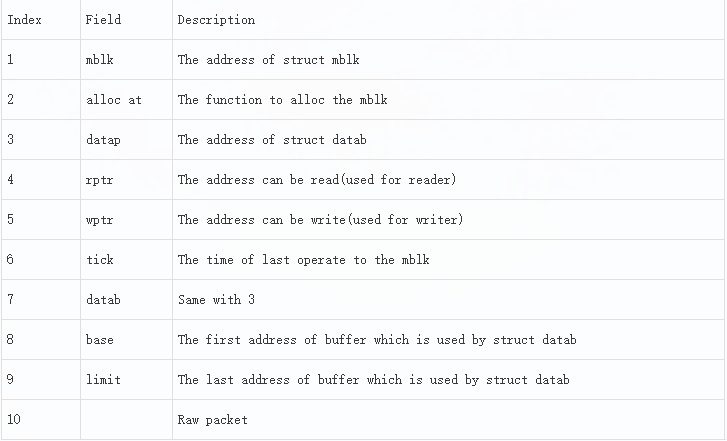
Print the value of mb in os\_demo\_mblk\_handle. show memory msgb.

The command “show memory <address>” can also be used.

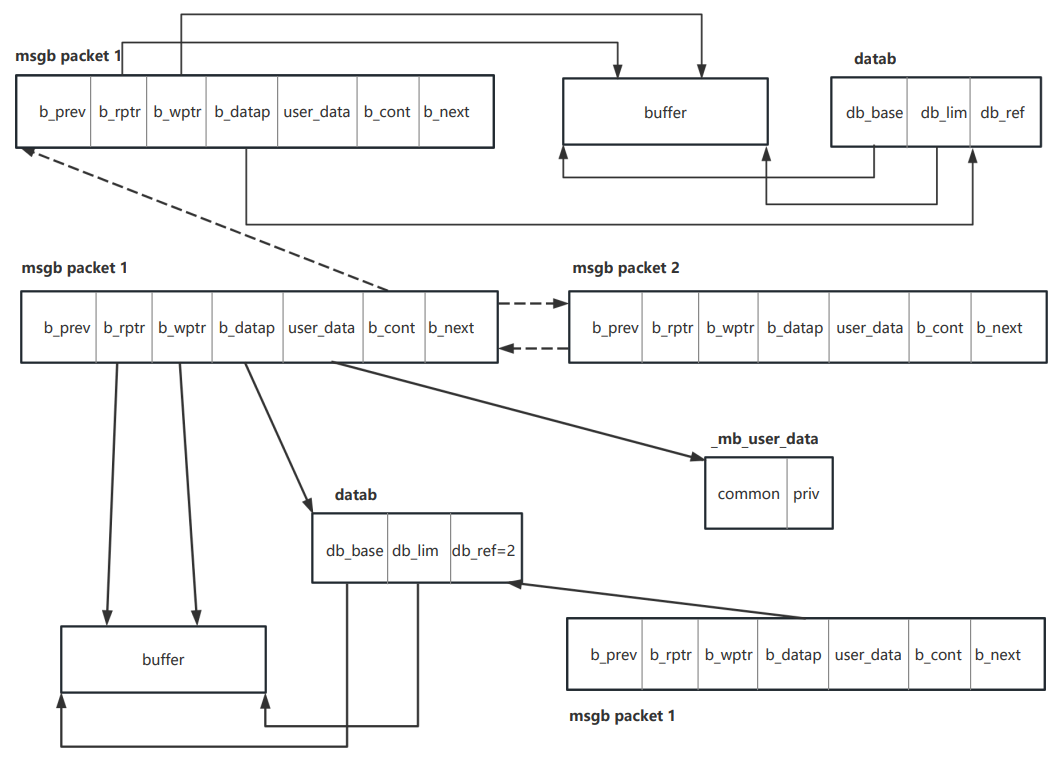
Question

1. Understand the meaning of all the fields displayed in the command "show memory msgb ", List them in a table.





1. Draw an image to show the relationship between msgb/datab and buffer.



### MBLK Leak

Experiment

To change the program "Receiving message, event, and multiple sockets in the same task," don't take the data away from the socket when the task receives the corresponding message from the message queue. Add a print when a message is received.

Observation and Thinking

1. To send the different sizes of data to switch, check the memory by the command "show memory mblk". What did you observe?
2. Do not operate the switch. Wait for one hour. What did you observe? What's the meaning of the output?

## Memory

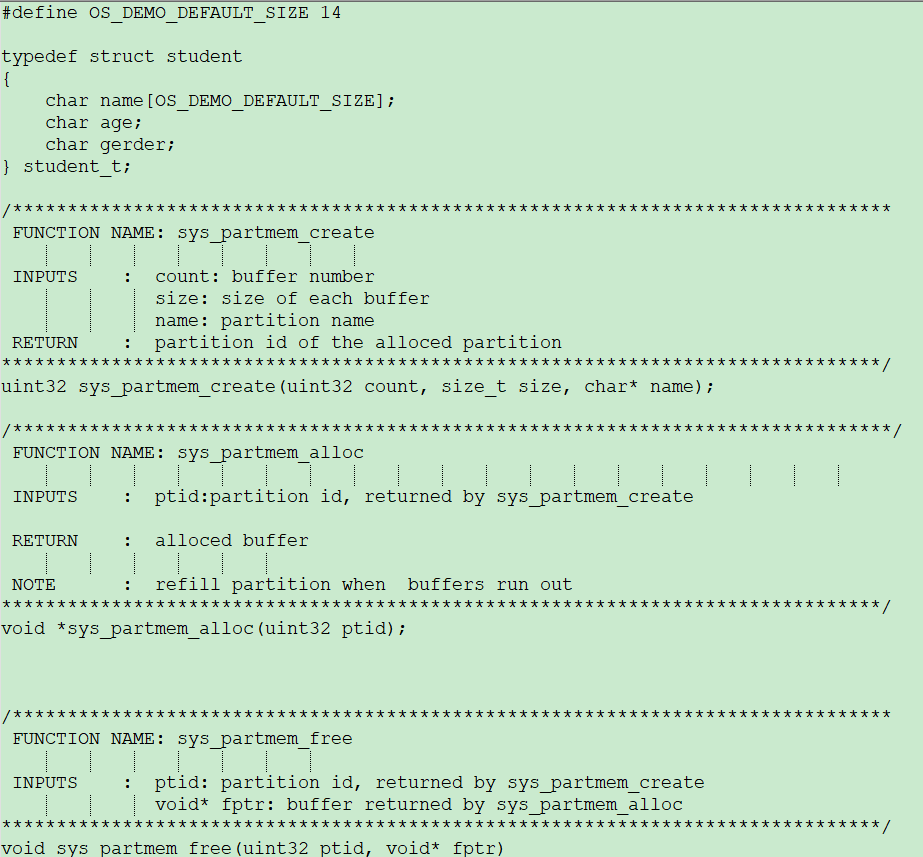
There are 2 types of dynamic memory in VxWorks. They are partition and region.

The partition is a memory area of a fixed length. Users can allocate buffers from the created partition. The features of the partition are: non-fragmentation, high efficiency, and waste.

A region is a memory region of variable length. Users can allocate segments from the created region. The features of the region are: fragmentation is easy, but it is flexible and does not waste.

### Partition Memory

* Programming
  + Create a partition memory whose count is 10 and size is student\_t.
  + Alloc the partition memory by the command "os-demo partition alloc"
  + Free specified block by command "os-demo partition free "
* Check
  + Check the partition memory by the command "show memory partition [block addr]"
  + Check the value of the range from start to end in a display of the command "show memory <memory addr>"
* Function and Structure



Questions

1. The total size we created for partition memory is 160 bytes, including 10 blocks \* 16 bytes for each block. The actual size allocated in the system is 240 bytes, get through end - start. What does the extra memory do?
2. Why does the command "show memory partition " display the location where the partition memory has already been deleted?

### Region Memory

* sys\_mem\_malloc/sys\_mem\_free
  + Allocate a new memory by the command "os-demo region alloc 200" through the function sys\_mem\_malloc; Find out this memory in the display of the command "show memory region 3"
  + Free the memory that you just allocated by the command "os-demo region free " through the function sys\_mem\_free; Check the state of this area's memory in the display of the command "show memory region 3"
* sys\_mem\_malloc/sys\_mem\_realloc
  + Allocate a new memory by the command "os-demo region alloc 200" through the function sys\_mem\_malloc; Find out this memory in the display of the command "show memory region 3"
  + Adjust the size of the memory that you just allocated by the command "os-demo region realloc 2000" through the function sys\_mem\_realloc; Check the state of the old memory in the display of the command "show memory region 3" and new memory in the display of the command "show memory region 6"
  + Adjust the size of the memory that you just allocated by the command "os-demo region realloc 20000" through the function sys\_mem\_realloc; Check the state of the old memory in the display of the command "show memory region 6" and the new memory in the display of the command "show memory heap detail"
* malloc/realloc/free
  + Using the C standard library functions malloc/realloc/free, replace the functions prefixed as sys\_mem. Do the experiment again.

Question

1. What did you learn after you finished this experiment?
2. What do you think needs to be paid attention to when using region memory?