

ITEE



## What's New:

The programming questions are designed to help you get in shape for doing assignments.

Learning objectives for this week:

- Understand the major concepts we've covered, including:
  - major computer components
  - major operating system types concepts
  - systems calls and I/O
  - basics of C programming
- Apply these concepts to reasoning about OS components and implementation, including understanding design choices.
- Understand the performance implications, including speed and usability, of major design decisions.

As before, you should aim to answer all of these questions by the end of the course. Being able to answer them by the end of week 3 is a useful goal. Be prepared: if you know what you can't do yourself, you will make most efficient use of the tutorial. The programming aspects are especially important as preparation for the assignments.

## 1. Operating System Structure

How does a virtual machine operating system differ from a microkernel operating system? Consider the following issues:

- a. Are the two incompatible concepts, i.e., can one have an operating system with features of both?  
*The 2 are different concepts but one could use a microkernel as a starting point to hide the hardware so that a VM system would be easier to design.*
- b. What are the differences in the interprocess communication models in the two kinds of OS?  
*In a microkernel system, IPC is via the kernel whereas in a VM system, you don't know about other processes in an extreme case where all processes have their own virtual machine, so communication is via virtual I/O devices.*
- c. Is the Java Virtual Machine any different to the general concept of a virtual machine OS? If so, how does it differ? If not, what are the major points of similarity?  
*JVM is oriented to supporting a specific programming language and may limit I/O access, as opposed to the general model, which gives an illusion of access to a whole system.*

## 2. Clients and Servers

In each of the following scenarios, which of machines A–E is the client and which is the server?

- a. In an X-windows environment, a program is running on machine A, displays windows on machine B, and a window manager on machine C controls the look and feel of the windows.  
*to A, B is a window server*  
*to B, A is an application server*  
*to A and B, C is a window manager server*  
*and in all cases, reverse the logic and you have the clients*
- b. In the example of part (a), a file is accessed on another computer on the network, machine D.  
*now all the servers are clients and the new machine is a server*
- c. To extend the example further, in the process of accessing the file on another machine on the network, the operating system has to check security information stored on a disk on machine E.  
*now E is a server, and to it, all the other machines are clients*

*The big-picture issue is that clients and servers are a software concept and it's often incorrect to label one machine as a client and another as a server: the relationship can change over time. Remember: client-server is a software concept.*

## 3. Mechanism vs. Policy

In a user interface manager, explain which of the following design principles are *mechanism*, and which are *policy*. In each case, comment on whether enforcing the design principle is or is not good for usability.

- a. Ability to draw a window on a different machine to that on which the requesting process is running.  
*Mechanism: this is something you can do. Whether good for usability depends how it's done.*
- b. Ability to change the language of text in the interface.  
*Mechanism. may be good for usability but depends how it's done.*
- c. Cut, copy and paste are always done using the same menu items and keyboard commands.  
*Policy. There is nothing about how this is implemented or how it's done: should be good for usability because consistency is easier than different behaviours.*
- d. Anything which can be drawn in a window can be printed.  
*Policy. Nothing is said about how it is done. This should aid usability because again the behaviour is consistent.*
- e. Although it is possible to move the mouse pointer under software control to an arbitrary location, its movement should always be under control of the user, through mouse movement.  
*Policy. This is a rule limiting what can be achieved (mechanism), presumably to aid usability.*

## 4. C programming

Start from the list code given in the lecture notes. Develop extensions as follows, noting which files they should go into (including headers and program files):

- a. Write a loop which visits each node in a list once, without modifying the existing data structures, and prints out the contents (assume all nodes contain a string)
 

```
ListHead test_list;
// assume the list is initialized
ListNode *current = test_list.first;
```

```

while (current) {
    printf ("loop : %s\n", current->contents);
    current = current->next;
}

```

- b. Extend the existing data structures to include a new struct, which stores a current location in list as well as a copy of the list head data structure, and can be used to keep track of the next location in the list. Call this new struct `Iterator`, and define a type for it using typedef.

```

// by storing a copy of the list not a pointer to the original, the iterator may miss
// changes -- not a robust solution because the internal pointers still point to the same
// place as the originals
typedef struct Iterator {
    ListHead the_list;
    ListNode * current;
} Iterator;

```

- c. Define functions which use the new `Iterator` data structure to:

- i. initialize the iterator with a new list

```

// in the header file:
void initIterator (Iterator *an_iterator, ListHead a_list);
void* getNext (Iterator *an_iterator);
int is_more (Iterator an_iterator);

```

```

// the function definitions: (detail of the rest by question)
// OK to copy the original list because we aren't modifying
// it but this design means that changes to the list will
// require a new copy of the iterator
// we can't use get_first because that returns the contents
// not a pointer to the first node
void initIterator (Iterator *an_iterator, ListHead a_list) {
    an_iterator->the_list = a_list;
    an_iterator->current = an_iterator->the_list.first;
}

```

- ii. get the next item from its list, moving the current item on (return NULL if past the end of the list)

```

void* getNext (Iterator *an_iterator) {
    void * return_value = NULL;
    if (an_iterator->current) {
        return_value = an_iterator->current->contents;
        an_iterator->current = an_iterator->current->next;
    }
    return return_value;
}

```

- iii. report whether there are more items in the list

```

int is_more (Iterator an_iterator) {
    return (an_iterator.current != NULL);
}

```

- d. Use the new type from part (c) to rewrite the loop of part (a) so it uses `Iterator`. Hint: you want code that looks something like this

```

while (is_more (list_loop))
    printf ("next word : %s\n", getNext (&list_loop));

```

That's pretty much the answer except you need to add in some initialization at the top:

```

Iterator list_loop;
initIterator (&list_loop, test_list);

```

- e. Comment on the relative efficiency and ease of use of the two loops you've developed: how does all this compare with an object-oriented language?

*The iterator version is easier to write: it requires less of the internal structure to be known in the rest of the program. Compared with an OO language, it's much harder to generalize and the lack of ability to hide details of data structures creates the temptation to rely*

on internal details in other parts of the program.

- f. Now consider how you would write code to deallocate an entire list. What would you do about deallocating the contents of the list?

*The difficulty is that you don't know how the contents was initially allocated. Code similar to the loop in the notes for `remove_head` could be used to deallocate the list nodes, but you don't have enough information to know whether the contents should also be deallocated. The only way to do this reliably is to store extra information with all dynamically allocated data showing how many pointers point to that data: a reference count. Everytime you make a new copy of a pointer, you increment the reference count. A simpler scheme is to pass in a flag when placing a data item on a list (or other dynamic structure) which says whether this structure is responsible for deallocating this data or not. A complication is that C has the `&` operation which allows a pointer to be created to data which was not dynamically allocated. This means you could (for example) put an item on a list which should never be explicitly deallocated. We will not explore this issue as fully as we should in this course for lack of time.*

**5. General: for discussion**

Small-scale systems with very small memories are coming back in the form of mobile and embedded devices. What challenges do you think these systems pose for operating systems which are similar to early developments of conventional operating systems (when memories were small, and processors slow)? What do you expect to be different?

*Some of the early mistakes have already been repeated, e.g., assuming that a proper OS won't be needed in early generations, then discovering this is an error as the sophistication of devices increases. 2 big differences are likely: real-time issues are big in these emerging areas, and many of these devices will only run specialized software (partly because they are so low-cost), so adapting to a completely new OS will be less of an issue. This last difference means that the trend in desktop and server systems towards fewer OS variations (anything that looks like Windows vs. anything that looks like UNIX) is less likely. Specialized OS designs to fit various niches is a more likely trend.*