CS 111 Scribe Notes

Lecture 4: OS organization

Presented: Wednesday, January 13th, 2016

This lecture picks up the concept of “Modularity” from last lecture and further divides it into two categories: soft modularity and hard modularity. From that, the lecture defines “Process” that utilizes hard modularity to enforce their independent share of computing resources.

From Last Lecture

* 1. Booting

Booting itself conforms to the concept of modularity. Firmware (BIOS or UEFI) is separated from disk or flash (identified by GUID: Globally Unique Identification). Pre-loaded booting program guides the OS to run via certain configurations. So the next question is how to enforce modularity after booting in OS? There are two options, hard modularity and soft modularity.

* 1. Soft modularity

Assume we want to write a function to implement fact.

int fact(int n) {

if(n) return 1;  
     return n\*fact(n-1);  
}

Or if we have already know the answer and store them in an array.

int fact(int n) {

return factable[n];  
 }

The x86 assembly code of fact() (with some added comments) is like follows:

fact:

pushl %ebp //push ebp onto stack

movl %esp,%ebp //ebp now point to top of stack

subl $16,%esp //allocate 16 bytes on top of stack

movl %edi,-4(%ebp) //store edi on stack(see L1 for restoring)

cmpl %0,-4(%ebp) //in case

jne .L2

movl $1,%eax //if n=0, return eax=1

jmp .L3

.L2

movl -4(%ebp),%eax //store old ebp

subl $1,%eax //eax = eax-1

movl %eax,%edi //store eax (new ebp for next loop)

call fact //loop

imull -4(%ebp),%eax //eax = eax \* ebp = eax \* eax+1

.L3

leave //bye

ret

For the leave function, esp=ebp, ebp=\*esp++;

The stack of fact() function can be understand as 5 steps like follows (figure 1).

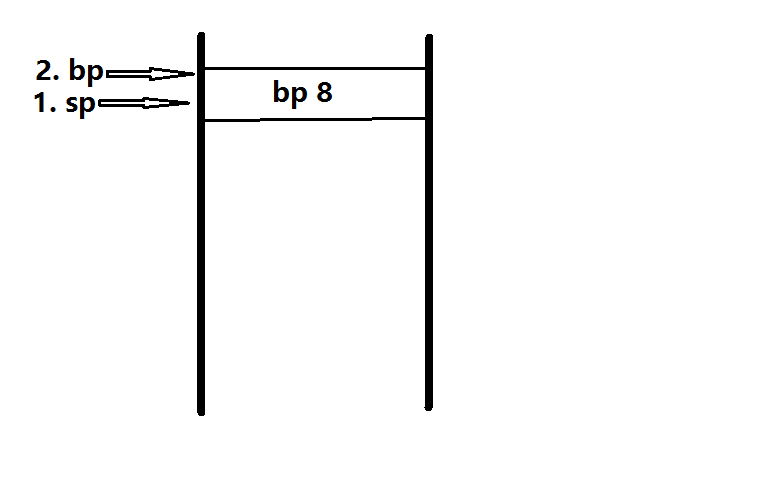


Figure 1



Figure 2

However, this code is inefficient.

Things can go wrong, for example, when we call:

fact(-1);

fact(50);

fact(INT\_MIN);

fact(1000000000000);

When we call fact(50), the bottom 32 bits will get incorrect answer. As for fact(INT\_MIN) and fact(1000000000000), the stack will keep going until the stack overflow. The way to solve it is to check every step to make sure there still some room left but will make the program slow. Another way is to use virtue memory.

To conclude, the soft modularity doesn’t scale to large apps.

To avoid this, we can use

1. client-server method, but it is slow;

2. virtualization. (Unidirectional hard modularity. Applications will be victims of kernel).

* 1. Hard modularity
     1. Simplest way

We can write a simulator for the machine that the applications run on. For this implementation, we should take care of certain instructions that cause halt and address out-of-range. We would also need to count instructions. When it reaches a limit, we would need to trigger a return and let the kernel judge which virtual machine that instruction should run.

All these things makes this simulated machine too slow to be put into production. Therefore we need a virtualizable processor to boost performance.

* + 1. Processor with virtualization

Idea: divide instructions into two classes. We need to trigger a protected transfer of control from application to kernel for privileged instructions. As such instructions are rare, this method would be efficient. As described in figure 3, figure 4.

Kernel

Application

n

Application

messes around (allowed)

Privileged ins

rip->mul

(request of protected memory)

unprivileged ins

Hardware

(not allowed)

mul

Figure 4

halt

Figure 3

* + 1. Classic way to enter the kernel
    2. Mechanism works:

Process is a program running in a virtual machine.

Creation:

Pid\_t fork(): clones current process. It returns childpid to parent, and returns -1 if failure, for example, no enough resources.

The following code is called fork bombs.

while(1) fork;

We call exit() function to stop running a process like follows:

void\_noreturn\_exit(int);

Waitpid() function is defined as follows:

pid\_t waitpid(pid\_t id, int status, in flags)