Read section 5.3.6 and 5.5.2 from SILBERSCHATZ and GALVIN.

We have already seen and implemented some scheduling policies namely FIFO, round-robin, priority, priority donation, etc. A user runs different kinds of applications, and sometimes they have conflicting needs. For example, an interactive application requires frequent scheduling; otherwise, the user may experience a delayed response. Applications who are doing frequent I/O want the earliest attention of the device. On the other hand, some applications are compute-intensive, and they can run in the background. A good OS scheduler must be able to identify these conflicting needs and implement a scheduling policy that makes everybody happy. One scheduling policy that is used by mainstream OS is multilevel feedback queue scheduling. A multilevel feedback queue scheduler maintains multiple queues of different priorities. A process is moved between these queues depending on how interactive the process is or how frequently the application is doing I/O. An interactive or I/O intensive process is given the highest priority. A compute-intensive application is given a lower priority.

The nature of a process can change over time, e.g., an I/O intensive application may become compute-intensive in future. Therefore, the scheduler needs to dynamically infer the behavior of a process and also change the priority accordingly. Because, an I/O intensive process is expected to yield shortly (while waiting for the device to respond), the scheduler uses a small-time quantum for high priority (I/O intensive) processes. For low priority (compute-intensive) processes a large time quantum is used.

If a process yields before the given time quantum or exhausts its time quantum, the priority of the process is changed accordingly. If a process yields before the time quantum then it must be doing some I/O and the scheduler tries to move to the even higher priority queue (with lesser time quantum). On the other hand, if a process exhausts its time quantum, then it must be doing some computation, and the scheduler moves it to the low priority queue.

One problem with priority scheduling is starvation. A low priority process may wait infinitely to get scheduled. To prevent starvation, the multi-queue feedback scheduler increases the priority of all processes which were not scheduled for a given period.

Processor affinity:

On multiprocessor hardware, the OS tries to schedule a process on the same core as much as possible. The private cache of a core might already contain data corresponding to a previously scheduled process that can be reused if the process gets scheduled again on the same core. If a process gets scheduled on a different core then all that data needs to be brought back again in the private cache of the target core. However, this optimization may not be effective in all cases. For example, if a core has nothing to do, it might schedule a process that was previously scheduled on a different core. Linux also provides system calls to set the processor affinity of a process or thread.

Now we are going to discuss some security concerns about the current OS model that we have discussed throughout this course. Most of the OS contains code related to file system, device drivers, memory management, network protocol, etc. Together these components sum up to a massive amount of code, and any vulnerability in this code would lead to serious security concerns (because everything runs in ring-0). In future lectures, we will try to discuss how to design an OS differently. In this lecture, let us focus on understanding, what kind of vulnerabilities exist in a C code.

To understand this, let us try to build a compiler that would never emit a privileged instruction (say deprivileged code). Assuming this compiler is correct, let us see if we can run a user application in ring-0.

One of the reasons why this can't be done because some instructions have different meanings in different rings. For example, a popf instruction restore the interrupt status flag from stack in ring-0 but not in ring-3. If we run an application in ring-0, it can disable interrupts by setting the interrupt flag to 0 using popf.

Let us assume that the instructions do not have different meanings in ring-3 and ring-0, can we still do a privileged operation while executing deprivileged code in ring-0.

The answer is yes. To understand that let us see a function pointer in C. A function pointer can be used to implement a callback. For example, a generic quicksort implementation which can sort and an arbitrary array of structures might not know how to compare two structure variables.

You can look at the qsort routine in stdlib.h. (run man qsort)

```
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

Here compar is a function pointer variable (callback) that contains the address of the function that can be used to compare two structures. The application code can use this generic quicksort implementation by defining their own compare functions and pass them as an argument to qsort. The address of the comparator function is stored in the argument variable compar. The application is allowed to modify the contents of compar variable. For example, compar++ will change the address stored in the compar variable.

```
foo:
b8 fa 00 00 00      mov $0xfa, %eax

main () {
   void (*fnptr)() = foo;
   fnptr++;
   fnptr ();
   return 0;
}
```

Let us look at the example above. If a function is dispatched using a function pointer (see fnptr() in the above example) the compiler emits instructions to load the address of the target routine from the function pointer and then emit an indirect call instruction to call the target routine. Notice, that in the above example the function pointer is further modified to store the address foo+1 instead of foo, and after the indirect jump the execution will start from the second byte of the foo routine. In other words, by modifying a function pointer, we can jump to the middle of an instruction.

Let us say the first instruction in the foo routine is "mov $0xfa, %eax". The byte code corresponding to this instruction is b8 fa 00 00 00. If the execution starts from the second byte, the hardware will see the opcode fa, which is the opcode of cli (interrupt disable) instruction. cli is a privileged instruction and the application will be able to execute it by modifying the function pointer even though the compiler didn't emit this instruction.

Corruption of function pointers is not the only way an application can execute arbitrary instruction. If an application accidentally makes an out of bound array access on the stack, it may overwrite the return address. In this case, after returning from the function, the execution may jump to arbitrary locations (including the middle of an instruction).