**2.3 Operating System Design and Implementation**

In this section, we discuss problems we face in designing and implementing an operating system. There are, of course, no complete solutions to such problems, but there are approaches that have proved successful.

**2.3.1 Design Goals**

The first problem in designing a system is to *define goals and specifications*. At the highest level, the design of the system will be affected by the choice of hardware and the type of system: batch, time shared, single user, multiuser, distributed, real time, or general purpose. Beyond this highest design level, the requirements may be much harder to specify. The requirements can, however, be divided into two basic groups: *user* goals and *system* goals.

*Users* desire certain obvious properties in a system. The system should be *convenient to use, easy to learn and to use, reliable, safe, and fast.* Of course, these specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them. A similar set of requirements can be defined by those people who must design, create, maintain, and operate the system. *The system should be easy to design, implement, and maintain; and it should be flexible, reliable, error free, and efficient.* Again, these requirements are vague and may be interpreted in various ways.

There is, in short, no unique solution to the problem of defining the requirements for an operating system. The wide range of systems in existence shows that different requirements can result in a large variety of solutions for different environments. For example, the requirements for VxWorks, a realtime operating system for embedded systems, must have been substantially different from those for MVS, a large multiuser, multi-access operating system for IBM mainframes.

Specifying and designing an operating system is a highly creative task. Although no textbook can tell you how to do it, general principles have been developed in the field of **software engineering**.

**2.3.2 Mechanisms and Policies** .,

One important principle is the separation of *policy**from mechanism*. *Mechanism* determine how to do something; *Policies* determine what will be done. For example, the timer construct is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.

The separation of policy and mechanism is important for flexibility. Policies are likely to change across places or over- time. In the worst case, each change in policy would require a change in the underlying mechanism. A general

mechanism insensitive to changes in policy would be more desirable. A change in policy would then require redefinition of only certain parameters of the system. For instance, consider a mechanism for giving priority to certain types of programs over others. If the mechanism, is properly separated from policy, it can be used either to support a policy decision that I/O-intensive programs should have priority over CPU-intensive ones or to support the opposite policy.

Policy decisions are important for all resource allocation. Whenever it is necessary to decide whether or not to allocate a resource, a policy decision must be made. Whenever the question is *how* rather than *what,* it is a mechanism that must be determined.

**2.3.3 Implementation**

Once an operating system is designed, it must be implemented. Traditionally, operating systems have been written in assembly language. Now, however, they are most commonly written in higher-level languages such as C or C++. The first system that was not written in assembly language was probably the Master Control Program (MCP) for Burroughs computers. MCP was written in a variant of ALGOL. MULTICS, developed at MIT, was written mainly in PL/1. The Linux and Windows XP operating systems are written mostly in C, although there are some small sections of assembly code for device drivers and for saving and restoring the state of registers.

The advantages of using a higher-level language, or at least a systems implementation language, for implementing operating systems are the same as those accrued when the language is used for application programs: *the code can be written faster, is more compact, and is easier to understand and debug. In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, an operating system is far easier to port-to move to some other hardware-if it is written in a higher-level language.* For example, MS-DOS was written in Intel 8088 assembly language. Consequently, it runs natively only on the Intel X86 family of CPUs. (Although MS-DOS runs natively only on Intel X86, emulators of the X86 instruction set allow the operating system to run non-natively slower, with more resource use-on other CPUs, are programs that duplicate the functionality of one system with another system.) The Linux operating system, in contrast, is written mostly in C and is available natively on a number of different CPUs, including Intel X86, Sun SPARC, and IBM PowerPC.

*The only possible disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements.* This, however is no longer a major issue in today's systems. Although an expert assembly-language programmer can produce efficient small routines, for large programs a modern compiler can perform complex analysis and apply sophisticated optimizations that produce excellent code. Modern processors have deep pipelining and multiple functional units that can handle the details of complex dependencies much more easily than can the human mind. As is true in other systems, major performance improvements in operating systems are more likely to be the result of better data structures and algorithms than of excellent assembly-language code. In addition, although operating systems are large, only a small amount of the code is critical to high performance; the memory manager and the CPU scheduler are probably the most critical routines. After the system is written and is working correctly, bottleneck routines can be identified and can be replaced with assembly-language equivalents.