A COMPUTING SYSTEM DESIGN FOR USER SERVICE

Webb T. Comfort
IBM Corporation
Yorktown Heights, New York

INTRODUCTION

After a long period of study and experimentation with various forms of user/terminal/system interaction, IBM is developing a general purpose time-sharing system. This is the recently announced System/360 Model 67 and the associated programming support package.

The basic technical objective of such a system is to provide a user at a console with what appears to him to be immediate response; i.e., when he asks for something relatively simple to be done, it should be done within 1 to 2 seconds. When he asks for difficult and complicated things to be done, there should not be an unreasonable amount of delay before they are in fact done. (This response time concept is very closely related to the current concern in batch processing systems over turnaround time.) Superimposed upon the response time requirement is the necessity to provide a broad scope of selectable procedures which allow a user at a console to simply and conveniently create, debug and execute his programs. More properly, he needs the necessary facilities at his fingertips to solve his problems.

However, from a marketing point of view, this is not sufficient. For those customers who have a requirement for a major facility of that type, the system must be able to support large numbers of such users simultaneously without an unreasonable amount of system overhead. Indeed, there are some computing installations in the country today who are prepared to take just such a step. On the other hand, there are also a good number of installations—in fact, probably the majority as of today—which have a requirement for some immediate access of this type, but not at the expense of crippling their normal batch processing operation. Consequently, the system design objective has to be to provide a flexible system which can provide either type of service (immediate access or standard batch) as the demand fluctuates.

One other point, which has been made increasingly clear in most of the pioneering time-sharing systems across the country, is that in the "hands-on" type of system operation being discussed here, long and arbitrary periods of system down time are simply unacceptable. As a result, the system design must include procedures for automatically handling as many hardware error situations as possible and avoiding a total system shutdown as long as possible.

Now, it is not the intent of this paper to describe in detail the Model 67 hardware, the programming system problems and solutions, or the specific user interface. Rather, some of the basic system charac-
A DESIGN FOR A MULTIPLE USER MULTIPROCESSING SYSTEM

characteristics will be discussed, particularly as they relate to the objectives pointed out above. Then the major hardware and software characteristics will be described. In this way, the reader should be able to get a feel for the overall system operation without getting bogged down in a myriad of details. Throughout the software discussions, the emphasis will be on the control program area rather than the language area, since it is the control program and its associated routines which determine how the system will operate.

It should also be noted that the system design has been influenced in a multitude of ways by various previous internal IBM efforts, including in particular the following:

- **TSM**¹
- **QUIKTRAN**², ³
- IBM's recently announced Administrative Terminal System
- Studies of One Level Store
- Studies of Polymorphic Multiprocessing

It has been aided by a number of discussions with various individuals from General Motors Corporation, Lincoln Laboratories, and particularly the University of Michigan.

**SYSTEM CHARACTERISTICS**

There are several basic characteristics of system operation which dictate how the hardware operates and what the design approach to the programming system must be.

The basic mode of operation is multiprogramming, wherein a multiplicity of tasks (represented by programs) reside in core at the same time and have access to common devices. However, the normal multiprogramming technique has been to run one task until it had to wait for some reason, such as for completion of some I/O. At that point, the CPU could be switched over to another task, to return to the first one later when its I/O was complete. Time-sharing goes a step beyond that, in that it is known that a certain (dynamic) number of tasks must be serviced within a reasonable period of time; namely, the response time mentioned in the Introduction. To this end, an algorithm is used to determine dynamically how much of the system's resources a task ought to be allowed to consume. If this threshold level is exceeded, the task is forced to stop and wait while other tasks have a chance; in other words, forced multiprogramming of sorts.⁴ (This has been generally termed as time-slicing.) In this way, a multiplicity of tasks—and therefore a multiplicity of users at consoles—can be accommodated.

The system is designed to operate with multiple CPU's and with multiple CPU-independent selection paths for I/O devices. This is necessary to provide the desired increase in system availability (as distinguished from reliability). It is also necessary in order to maintain orderly growth, particularly as it will probably not be possible to specify optimal system configurations until after the system has been in operation for a while. In addition, this allows for partitioning within the system, to provide smaller but independent subsystems, if desired. (The same system will, of course, operate with only one CPU.)

Dynamic relocation is implemented (as described in the next section) and applied. This allows a task to operate as if it had a full addressable memory (called a virtual memory) of \(2^{34}\) (or, in the larger CPU, \(2^{35}\)) bytes, almost independent of the amount of real core provided in the system. Dynamic relocation can be used to simplify the traditional relocation techniques. This is particularly important in time-sharing, where it often becomes necessary to throw a task out of real core before it is finished and bring it back later (the forced multiprogramming indicated above). Dynamic relocation completely eliminates the problem of having to return the task to the identical area of core it occupied during its previous period of residency.

There are two characteristics of program execution which have heretofore been unexploitable. The first is that in general a program must claim an amount of core large enough for its worst case operation, even though in many cases, either during debugging or based upon parameter values or program structure, the actual core requirement for a given execution is significantly less than that maximal amount. The second is that in many cases program activity is localized for significant periods of time, i.e., during its execution, a certain set of routines will run for a while, and then another set, while the first

---

¹In many places in this paper, for purposes of avoiding unwarranted complexities, general statements are made. Many of them, if interpreted literally and assumed applicable in all cases, would be clearly objectionable. It must be emphasized that such generalities are for expository purposes only, but do represent the basic system action.
are not used. To capitalize on these situations, the **paging** concept is used, wherein through the dynamic relocation facility the control program can recognize on a dynamic basis which parts of a program or its data are now required, and which are no longer required. Virtual memory is then broken up into blocks called pages, and only those pages actually referenced by the task will be brought into real core. This will provide a better utilization of real core storage, as well as allowing an equivalent level of multiprogramming operation (when compared with always moving a complete package of a program and its data) but within a significantly smaller amount of real core.

In the past, it has generally been necessary to declare at least by load time all subroutines which might possibly be needed during execution, and to load them all into memory before execution can begin. Dynamic linkage (or, more properly, dynamic loading) is a facility whereby a routine need not be declared or loaded until, at execution time, it is actually called. (At call time, virtual memory can be allocated for the called routine and external symbol definitions resolved. Actual relocation modification of addresses—or, in System/360 terms, address constants—need not occur until paging time.) This is particularly important for a user at a console, for he may elect in the middle of an execution to change his mind about what subroutine he wants the program to call—and he ought not to be required to stop and reassemble or reload. This also contributes to more effective utilization of real core.

A general purpose terminal oriented system would not be complete without a “warehouse” of previously stored programs and data sets, maintained and cataloged by the system, and callable by the user or his program.

The scope of user facilities is defined by the set of languages at his command. The basic Model 67 program package will include a Command Language (with a set of Program Checkout techniques), a FORTRAN compiler, and a macro assembler. Later extensions will include COBOL and PL/1.

**HARDWARE CHARACTERISTICS**

Several extensions and modifications have been made to System/360 hardware in order to facilitate the system characteristics summarized in the Systems Characteristics section. The most important of these will be discussed briefly here. (It is assumed that the reader is familiar with the basic System/360.4) In those situations where the CPU is altered, there is generally a switch of some sort (physical or programmable) which will disable the feature, thus assuring that Model 67 will still run programs prepared for any other model of System/360.

Standard System/360 has a 32-bit word, and 24-bit addressability; i.e., $2^{32}-1$ is the largest memory address recognizable by the hardware. On the Model 67, an option is provided for full 32-bit addressing. Associated with this facility is a new instruction, Branch And Link, which is essentially a 32-bit version of the Branch And Link instruction.

In System/360, an effective address is formed in the general case by forming the sum of the contents of a base register, the contents of an index register and the contents of the displacement field specified in the instruction. The Model 67 is provided with a program-controlled relocation mode, which imposes a translation function between the time the logical (effective) address is generated and the time the address is sent to the appropriate memory box. The total addressing space (virtual memory) is broken up into pages of 4096 bytes apiece. Thus a logical address can be considered to be a 12-bit page number (optionally 20-bits with 32-bit addressability) concatenated with a 12-bit byte address within the page. The basic mechanism is to provide a relocation table for a direct look-up of the logical page number, and to fetch a new physical page number from the table. Because of its potential size, this relocation table is kept in main core storage rather than in its own hardware implementation. However, since the translation must occur on the fetch of every instruction and operand from core storage, a set of associative registers are provided to reduce the number of additional memory references to an acceptable level.

Figure 1 shows a simplified data flow for the relocation hardware. It will be noted that the page number translation is a two-step process. That is, pages are divided into groups of 256, called segments. The logical segment number is looked up in a segment table, and determines the location of a page table, which is then used to translate the logical page number into a physical page number.
Availability bits are provided at both levels, and are used to cause CPU interrupts when references are made during execution to selected segments or pages. (This is how the paging technique is implemented.) This two-level relocation technique provides the following:

1. A convenient way to allocate a data area of unknown length, by allocating it at the beginning of a segment, and allowing it to fill the segment.
2. A way of compressing page tables, since actual table entries are not required for unused pages at the end of a segment.
3. A way of reducing the amount of real core required to contain page tables at execution time, since segments can be marked unavailable, and the associated referencing interrupt used as a signal to bring that particular page table into core.
4. A very convenient way of sharing programs or data sets among tasks, since a one-page table could be pointed to by segment table entries for several different tasks.
5. An effective mechanism for read/write protection of areas of virtual memory. The reason for a segment being marked unavailable is open to interpretation by the control program.
6. A simple and efficient overlay mechanism, through substitution of segment table entries.

Associated with the relocation feature is a new privileged instruction, Load Real Address, which allows the control program to find the translated form of any virtual memory (logical) address.

In System/360, basic CPU control is contained with a Program Status Word (PSW). In Model 67,
this PSW control has been extended by a set of con-
trol registers, whose contents include the table regis-
ter (which defines where the active segment table
is), extended masking facilities, and program-read-
able status indicators for various system switches.
Associated with this capability is a pair of new in-
structions, Load Multiple Control and Store Multi-
ple Control, which allow for manipulation of the
control register contents.

In order to allow for multiple-CPU operation,
multiple memory bus connections are provided.
Two-channel switches and channel controllers serve
to provide CPU-independent selection and control
of I/O. Extended Direct Control provides for inter-
CPU communication. The instruction Test And Set
allows guaranteed interlocking where required.

Partitioning switches are included at critical
communication points within the hardware to allow
for dynamic manipulation of the hardware system
configuration. New devices can be added, others
removed for testing, or a full subsystem partitioned
out for independent operation. This facility must be
carefully controlled in actual operation to prevent
unintentional system operator slips from disrupting
the system. On the other hand, ultimate authority
must rest with a human being, to safeguard against
undebugged or recalcitrant programs.

SOFTWARE CHARACTERISTICS

The basic unit of control in the system is a task.
For a user at a console, one task is defined to pro-
vide services for his complete session at the termi-
nal (i.e., from logging on to logging off). For a
non-conversational (batch) job, a task controls the
reading and interpretation of the job control cards
and requested services.

Each task operates within its own Virtual memo-
ry; i.e., there is a set of relocation tables for each
task. When multiprogramming among tasks, simply
changing the table register causes a new set of relo-
cation tables to be brought into use.

When looked at from the point of view of one
task, the software can be thought of as having three
levels as pictured in Fig. 2. There is a basic system
Supervisor, which has the following characteristics:

![Diagram](image)

Figure 2. The software world, as seen by one task.

- Permanently resident in real core
- Runs non-relocated
- Not addressable in Virtual Memory
- Runs in Supervisor State
- Not time-sliced or paged

This is the Supervisor program which is common
to all tasks and all CPU's. It handles all the details
associated with interrupts, I/O, paging and sched-
uling. Figure 3 shows the major pieces of the Su-
ervisor, as it is now defined. Without discussing
each one, a few comments will clarify the opera-
tion:

1. The Supervisor is basically an interrupt
handler. System 360 hardware automatically
sorts out five types of interrupts (called
I/O, Program, SVC, External and Machine
Check). The Supervisor must sort each of
these types into finer categories and cause
the appropriate actions. As a general rule
of operation, interrupts are stacked by the
Supervisor; i.e., if the Supervisor is pro-
cessing one interrupt when a second one occurs, the second one will be put on a queue until processing of the first is complete. All such pending interrupts will be processed (if possible) before returning to one of the tasks in Virtual Memory.

2. Unfortunately, there are many situations within the Supervisor where requests are made for services or facilities which are busy on other things. Whenever this situation occurs, it is necessary to form a queue of some sort. In order to handle queued requests in a reasonably uniform way, all such queues are controlled at a central point. There is a queue provided for each type of hardware interrupt (which is how interrupts are stacked), as well as a queue for each facility (such as real core storage, drum space, and I/O device use) which might be busy when a request is made. A generalized queue scanner is then provided to see to it that whenever any CPU is in the Supervisor, it will handle all outstanding serviceable queued items.

3. The major portion of what is generally called the scheduling algorithm is built into the Dispatcher. Since everyone seems to have his own idea about what is a good scheduling algorithm, its most important characteristic in this system is to make it as modular as possible so as to be easily changeable, or even replaceable. Rather than outline a specific technique, it will suffice here to list the conflicting objectives which such an algorithm must attempt to satisfy.

(a) Provide "reasonable" response at a terminal to a request for a "reasonable" amount of work
(b) Provide good throughput on batch-type, non-conversational tasks.

Figure 3. Supervisor block diagram.
A COMPUTING SYSTEM DESIGN FOR USER SERVICE

(c) Provide some degree of balance between (a) and (b), based upon relative loads.
(d) Utilize the paging concept as efficiently as possible.
(e) Implement multiprogramming of time-shared tasks, as required by (d).
(f) Provide dynamic variation of (d) and (e) to match CPU speed with drum and disk rates.
(g) Provide some correlation with a user-oriented priority scheme.
(h) Be simple and fast.

4. When a CPU recognizes a hardware error (i.e., a Machine Check Interrupt occurs), that CPU goes into Wait State, and another CPU is alerted (via Malfunction Alert Interrupt through Direct Control). The alerted CPU then invokes a recovery procedure.

5. Because the system is designed to operate by paging, such paging I/O functions are handled separately, to make them as efficient as possible.

6. Pathfinding is the general control mechanism for I/O selection and utilization.

Within a task's Virtual Memory are a set of Service Routines, which are a part of the programming support, and which provide in effect a one-task supervisor to control the services and communications for the user with whom this task is associated. These routines will in general be reentrant, and shared among all tasks. Some of these service routines are defined to operate with a privileged action capability (which is not the same as the Supervisor State recognized by the System/360 hardware). Such routines are allowed to request special actions by the Supervisor (such as changing relocation tables), and are protected from the rest of the programs (including those of the user) within Virtual Memory. Such privileged service routines include the following:

1. The program which interprets terminal commands.
2. Some of the subprograms which carry out terminal commands.
3. All services relative to the catalog (searching, interrogating, changing, etc.)
4. Virtual Memory allocation.
5. Private device allocation.
6. Allocation of external (catalogable) space.
7. Access methods, which provide the task with GET/PUT capability to terminals, sequential, and direct access devices.
8. Dynamic loader, and associated tables.
9. Mechanism for handling task interrupts (as distinguished from hardware interrupts).

The nonprivileged area of Virtual Memory (which is by far the bulk of it) is available for the language processors (FORTRAN, PL/1, etc.) and user-generated routines.

The basis behind the three levels depicted in Fig. 2 is protection. The Supervisor is protected from accidental random damage by virtue of the fact that it is not addressable in anyone's Virtual Memory. (What is addressable in a task's Virtual Memory is determined by what the Supervisor puts into the page table for the task. Note that this also provides complete protection between different tasks.) Communication with the Supervisor is limited to a specific set of SVC's (Supervisor Calls, a System/360 interrupt mechanism for that specific purpose). However, the majority of such possible requests are of a very sensitive nature, i.e., if misused, they could seriously affect the operation of the whole system, and this is highly undesirable. To help control this problem, the set of privileged service routines was defined, and execution of the sensitive requests is limited to such service routines. In turn, access to these privileged service routines is limited to legitimate entry points, and they are protected (via System/360 protection keys) from access by non-privileged routines. In this way, it is hoped to eliminate the possibility that an undebugged or irresponsible task could hurt anyone but itself.

SUMMARY

This report attempts to give an over-all picture of the System/360 Model 67 Time Sharing System, its system design, and major hardware and control program characteristics. The unique combination of hardware and software objectives makes a very complex problem, for which a simple and efficient solution is desired—a difficult task at best. It is further complicated by the fact that there is no recognized consistent way to measure such a system, either in how it performs or how well specific technical problems have been solved.
It is the author's opinion that one very significant concept made available in this system is the large addressable Virtual Memory. It should force a complete reevaluation of how programs should be written, and has the potentiality of making obsolete the traditional I/O techniques. However, it will require a good deal of experience and experimentation to know how best to exploit this new facility.

REFERENCES


