ON THE DEVELOPMENT EFFORT OF AN
INSTRUCTIONAL OPERATING SYSTEM PROJECT

Manfred Ruschitzka

DCS-TR-65

Rutgers University
November 1977

This paper was submitted for publication in the Proceedings of the Third International Conference on Software Engineering in Atlanta, Georgia, 1978. Author's address: Department of Computer Science, Rutgers University, New Brunswick, NJ 08903.
ABSTRACT

The undergraduate course on operating systems at Rutgers University is coupled with an implementation project. To serve its purpose, such a project must incorporate many features of contemporary operating systems, but it must also be manageable within the given timeframe. We describe the nature of this project and the adopted development methodology which simplified the construction phase by increasing the complexity of the analysis and the design. Statistics about the development effort, student productivity, and computing resource usage are presented and related to those of industrial efforts. The high productivity rate is attributed to the development methodology which is not restricted to instructional software projects.
INTRODUCTION

In the spring of 1975 we introduced an implementation project in our senior-level course on operating systems at Rutgers University. Our goal was to make this project as realistic as the given time frame of one semester would allow. Thus, much of the project analysis was concerned about reducing the complexity of the implementation effort. Several iterations of top-down and bottom-up design steps permitted us to locate those modules which appeared unnecessarily time-consuming to implement. Often, their complexity could be reduced by providing appropriate support primitives. The resulting system architecture was both cleaner and less complex, thus keeping the implementation effort manageable.

Almost all students who enrolled in the course during the last three years successfully completed the project within one semester. While this certainly proved that the effort was manageable, we lacked quantitative measures for its evaluation. In the spring of 1977 we therefore asked students to keep logbooks on the progress of their software development. The statistics reported in this paper are based on the logbooks and the accounting data provided by the computer center.

The project involves both a hardware simulator and an operating system. There are two compatible hardware models (COS-1 [5] and COS-2 [6]) which aid in the incremental
development of the simulator. The multiprogramming operating system which is run on this simulator is termed COSMOS [8] and written in CAL [7], the high-level flavored COS assembly language. To convey the scope of the project we precede the discussion of our statistics by a characterization of the system components. For more details, as well as the motivations behind the choice of the various architectural features, the reader is referred to [9].

THE HARDWARE SIMULATION

The configuration of the simulated COS-2 hardware complex is illustrated in Figure 1. The CPU contains a paging map and eight registers: program counter, instruction register, accumulator, index register, interrupt and trap register, arming register, mode register, and state pointer register. The latter contains a pointer to the process state record (PSR, the data structure for the state of a process) of the currently running process. Instructions have a fixed one-word format, contain two address fields, and permit four addressing modes. The set of about 60 instructions contains special operating system instructions for switching the state of the processor between processes (SWITCH PROCESS), bypassing the paging map, manipulating bit-tables, and chaining queue records. The operating system call instruction (OS CALL) is implemented as a trap.
Figure 1. COS-2 system configuration.
The virtual address space consists of 8 pages of 128 words each. Primary memory consists of 2K words or 16 pages. Page 0 contains a number of standard locations ("low core") for bootstrapping, timers, the drum queue pointer, and the interrupt transfer vector. The drum system has a storage capacity of 16 pages. Transfers are on a page basis and are controlled by the drum controller which obtains the transfer parameters from the drum transfer record queue in low core.

The lineprinter can be activated by two instructions. PRINT LINE prints the contents of the accumulator as an eight-digit hexadecimal integer and advances one line. A special purpose instruction, HEADER, is provided to print the header preceding the output of a user job. HEADER prints several lines of symbolic and numerical information which is extracted from a process state record. The information contains job statistics and possibly indications of an abnormal termination.

The cardreader serves as the system input device. If programs were entered via the cardreader in symbolic form, they would require about 20 times more storage space than machine code. This is avoided by coupling the cardreader with a crossassembler (CAROL [7]). CAROL is written as a subroutine which, when called, returns the next object word of the previously assembled job as well as an indication about the type of the corresponding card (instruction,
control card, etc.). If no more object words are available, the next job (format: JOB card, program, [DATA card, input data,] END card) is assembled and then the first word (the JOB card equivalent) is returned. The instruction READ CARD is implemented as a call to CAROL and skips a variable number of times depending on the type of card.

The console features a breakpoint facility which aids in debugging system software. There is also a bootstrap mechanism for loading stand-alone programs from the cardreader. In a simulation run, the input data to the simulator (or, more specifically, CAROL) consist of the symbolic COSMOS code (which will be assembled and bootstrapped) followed by an arbitrary number of user jobs (to be spooled in by COSMOS).

THE COSMOS OPERATING SYSTEM

COSMOS is brought up by bootstrapping from the cardreader. After initialization, it accepts user jobs from the cardreader for processing. Syntactically incorrect jobs are flushed by the crossassembler and are not passed on to the cardreader. At any one time, several jobs may be in the system at various stages of completion. Any one job, however, will be processed in the following sequence. After being spooled to the drum, a job's program is loaded into core and then given control of the CPU. For input, the program may execute an OS CALL which will provide it with
the next input word from the input data page on the drum. Another OS CALL causes the transfer of an output word to the output data page on the drum. After termination of a user job, its output is printed on the lineprinter together with some statistics. Because of this default sequence there is no explicit command processor.

COSMOS is structured in levels of abstraction as shown in Figure 2. The mode of operation determined the high-level functions of spooling, I/O, loading, termination, and printing. The concept of multiprogramming is implemented by the SCHEDULER one level below so that the high-level functions can be scheduled like user processes. The interrupt routines in level 4 provide real-time independence for all higher levels. Drum-multiprogramming, realized by entering drum transfer records in the drum queue, is real-time dependent and thus one level below. The process notion is represented by the process state records which are allocated to level 2. Finally, data structures must be allocated before they can be manipulated. Thus, storage allocation has been assigned to the lowest level in the hierarchy. All modules in levels 1 and 2 are implemented as instructions, thus simplifying and compacting the COSMOS code. PUTDRQ on level 3 is implemented as a shared (and interruptible) subroutine to illustrate synchronization problems. All modules above level 3 are implemented as processes.
<table>
<thead>
<tr>
<th>LEVEL</th>
<th>ABSTRACTIONS</th>
<th>FUNCTIONS</th>
<th>MODULES</th>
<th>DATA STR.</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td>user problem execution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>virtual user memory, files, OS CALL's</td>
<td>user support (spooling, I/O, loading, termination, printing)</td>
<td>RCARD, WCARD, LOADER, FILE SYSTEM, TERMINATOR, PRINT</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>CPU multi-programming</td>
<td>CPU allocation</td>
<td>SCHEDULER</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>real-time independence</td>
<td>service of interrupts and traps</td>
<td>Timer Interrupt Process, Drum Interrupts Process, Trap Process</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>drum multi-programming</td>
<td>allocation of drum controller</td>
<td>PUTDRQ</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CPU and drum processes</td>
<td>accesses of PSR's and DTR's</td>
<td>Queue instructions, SWITCH PROCESS, START DRUM</td>
<td>PSR queue, drum queue</td>
</tr>
<tr>
<td>1</td>
<td>storage allocation</td>
<td>accesses of allocation words</td>
<td>ALLOCATE, DEALLOCATE</td>
<td>core and drum allocation words</td>
</tr>
</tbody>
</table>

Figure 2. Levels of abstraction in COSMOS.
Control is transferred between two processes by executing a SWITCH PROCESS instruction or by invoking the interrupt system (the simulator uses the same subroutine for both). User processes relinquish control by executing an OS CALL which causes a trap. The task of activating, deactivating, blocking, and unblocking processes, accomplished by setting and/or resetting bits in the status words of the corresponding P3R's, is distributed among the COSMOS processes. For example, since loading of a job depends on the successful completion of the spooling operation, the spooling process which finishes this operation will activate the loader on behalf of this job. After system initialization, the SCHEDULER loops scanning the process state record queue and picking the ready job with the highest priority for execution. Subsequently, a SWITCH PROCESS instruction transfers control to the selected process. If no processes are ready, the SCHEDULER checks whether there are any blocked ones; if not, the simulation run is terminated.

The Drum Interrupt Process cleans up after a transfer and restarts the drum if more requests are pending. It also supports the FILE SYSTEM in performing user I/O. The Trap Process handles OS CALL's as well as protection violations. User programs exceeding their time limits (set by the SCHEDULER) lose control to the Timer Interrupt Process when the timer goes off.
A double-buffering scheme with dynamic buffer allocation is used for spooling jobs from the cardreader to the drum. The two processes RCARD and WCARD read a page into core and write it to the drum respectively. They synchronize each other and also initialize a PSR for the arriving job. After a job has been spooled in, the LOADER will transfer its code to primary memory as soon as there are enough pages available. User jobs read input data and write output data sequentially by executing the appropriate OS CALL's which in turn cause an activation of the FILE SYSTEM. Buffers required for the input or output page are allocated dynamically and their locations as well as the access indices are maintained in the PSR of the requesting user. The end of execution of a user job is signaled to COSMOS by another OS CALL. The TERMINATOR will subsequently release all allocated resources except for the output data file. Jobs may also terminate abnormally by executing an illegal instruction, by causing a memory violation, or by exceeding their quanta. A user job's output is preceded by statistics and printed by the PRINT process which finally returns the job's PSR to the freelist.

**DESIGN METHODOLOGY**

To serve its purpose, the simulated system had to be as realistic as the time limit of one semester would allow. Thus, the minimization of the cost/benefit ratio was our primary criterion in the definition of the operational and
the systems requirements [9]. In this spirit, we defined both the COS configuration and the COSMOS operating system as representative of contemporary systems, but more modest in scope and not based on any particular brand. As a compromise, we decided that a team of two students should implement the entire system. More students per team would allow a more complex system and emphasize software engineering aspects, but the individual student would then not be able to absorb all the implementation details. Conversely, if a single student were to implement the entire system, the excessive programming effort would leave little time for conceptual and analytical aspects.

How much code could a team of two students be expected to produce within one semester? Due to the lack of published data on student programmer productivity, we first used industrial data which indicate that a system programmer produces between 0.24 and 12 machine instructions/hour [2]. On the average, this works out as one instruction/hour [2,3,10]. This rate includes the analysis, the design, coding, debugging, system testing, and documentation. Assuming a 40-hour week and a 15-week semester, a team of 2 students registered for a normal load of 4 courses would devote 40*15*2/4=300 hours and produce an equal number of instructions. There are at least three major reasons why actual student productivity rates tend to exceed this value. First, the measure of machine instructions/hour (or instructions/man-month) evolved at a time when industrial
real-time programs were predominantly coded in assembly language. There is now reason to believe that programmer productivity is invariant to the type of language when measured in source statements/hour [c.f. 11]. The use of a high-level language would therefore increase productivity by the expansion factor of the particular language. Second, there is quite a difference in effort between developing a functioning program, a maintainable and documented programming product, an integrated programming system, and a programming systems product which has all of these characteristics. Brooks [1] found that a programming systems product devours nine times the effort of an equivalent program. While we felt that it was important to spend the effort required for a programming product, the integration effort of a team of two does not pose an appreciable additional effort. Third, not all of the activities reflected in the infamous 1 instruction/hour are required of a student programmer. In a course project, the instructor provides much of the analysis and often well-defined design and even coding specifications.

Since the design effort is very much a function of the quality of the design specifications, we decided to provide students with well-defined and sometimes quite detailed documents of reference manual quality. To arrive at these documents, we performed several iterations of both top-down and bottom-up design steps. These iterations started with the definition of the mode of systems operation, followed by
a sketch of the hardware configuration and the higher levels of abstraction of the operating system. We then proceeded to code the operating system in a hypothetical, open-ended assembly language. After a frequency analysis, superfluous instructions were deleted and special purpose instructions were added to replace frequent instruction sequences and functions which were incompatible with the evolving architecture. This methodology of integrated design allowed us to move functions from hardware to software and vice versa. Wordsize and instruction format were not fixed until the code of a rudimentary version of the operating system had been sketched out. Eventually, the initial general-purpose processor had evolved into a special-purpose processor with appropriate operating system support features.

During the last three years almost all student teams finished the entire system within one semester. A small increase in the required effort, however, would have resulted in a much higher drop-out rate. While much more modest in scope, the project resembles large industrial software developments with respect to the necessity of completing a complex task within a limited amount of time. It appears that this methodology could be generalized for large industrial efforts. In essence, a system should be thoroughly analyzed and partially implemented before the design specifications for hardware, microcode, and software modules are frozen. By providing explicit support for
higher level modules, this approach reduces the software management problems which are usually the most complex problems in the development of a system.

**PROJECT ORGANIZATION**

In the spring of 1977 the project was implemented on an IBM 370/158-168 complex. Programs were developed interactively in CALL-OS on the 158 and submitted from the terminal for batch-processing on the 168. Half the class chose FORTRAN for the implementation of the COS simulator, the other half used PL/I. The crossassembler was written in PL/I by a teaching assistant and two loadmodules (with different interfaces for FORTRAN and PL/I) were made available. COSMOS was to be written in CAL, the assembly language of COS.

There are six steps in the project, and each step builds on all preceding ones:

1. Simulation and testing of COS-1.
2. Simulation and testing of COS-2.
3. Run of a bootstrapped periphery test program.
4. Testing of the COSMOS spooling system.
5. Run of a single user job under COSMOS.
6. Run and evaluation of a batch of user jobs.

The completion of a step corresponded to a milestone. Penalty points were subtracted for failure to meet milestone deadlines. Four reference manuals, one each for COS-1 [5], COS-2 [6], CAL-CAROL [7] and COSMOS [8], provided the design
specifications. Less detailed coding specifications were given for each of the six steps. Depending on the complexity of a step, deadlines were spaced at intervals ranging from one to two weeks. To reach a milestone, the two students in a team had to design, code, debug, and document several modules. In addition to in-line documentation, hand-written comments on the operating system activities during a simulation run were required on the final printout and the logbook entries on programming activities and computer usage had to be current.

The statistics in the next section have been derived from entries in these logbooks. Computer usage data were then compared with the accounting data supplied by the computer center. With the exception of connect time data which were affected by a bug in the accounting system, discrepancies were generally below 20%. The logbooks also contain entries for the time spent on design, coding, typing, debugging, and documentation.

**PROJECT STATISTICS**

Twenty-eight students undertook the project in the spring semester of 1977. The average times which a student spent on the various project activities are summarized in Table I.
<table>
<thead>
<tr>
<th>Activity</th>
<th>Hours</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class meetings (design)</td>
<td>39.0</td>
<td>23.9</td>
</tr>
<tr>
<td>Studying of design</td>
<td>16.0</td>
<td>9.8</td>
</tr>
<tr>
<td>Coding</td>
<td>19.4</td>
<td>11.9</td>
</tr>
<tr>
<td>Debugging</td>
<td>50.1</td>
<td>30.8</td>
</tr>
<tr>
<td>Documentation</td>
<td>19.4</td>
<td>11.9</td>
</tr>
<tr>
<td>Typing</td>
<td>19.0</td>
<td>11.7</td>
</tr>
<tr>
<td><strong>Total effort</strong></td>
<td><strong>152.9</strong></td>
<td><strong>100.0</strong></td>
</tr>
</tbody>
</table>

Table I: Distribution of COSMOS effort by activity.

It is instructive to compare these data with industrial averages which are often estimated by the 40-20-40 rule. This rule of thumb states that analysis and design account for 40%, coding and debugging for 20%, and system testing for 40% of the total effort (in terms of cost or man-months). Documentation is assumed to be part of the various activities, but the most time-consuming final documentation is lumped in with system testing. Several researchers have independently discovered this rule; we shall use Metzelaar's data [10, Table III] because of their finer subdivision of activities. Metzelaar's data are depicted in Figure 3. Note that neither analysis (20%) nor system testing (28.3%) were required in our student project; the former was performed by the instructor and the latter did not apply because of the nature of the project and the
Figure 3. Distribution of a typical large industrial software development effort by activity (Metzelaar).

Figure 4. Normalized distribution of the COSMOS student effort by activity.
small team size. Thus, according to Metzelaar's data which represent the 40-20-40 rule, students performed only 51.7% of the effort of a comparable industrial software development project. In order to relate our data to these industrial ones, we merge the first two, the second two, and the third two activities in Table I and call the new categories design, coding and debugging, and documentation respectively. In Table II, these new activity categories and their normalized percentages (which add up to 51.7%) are listed together with Metzelaar's industrial averages.

<table>
<thead>
<tr>
<th>Activity</th>
<th>% COSMOS</th>
<th>% Industry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis</td>
<td></td>
<td>20.0</td>
</tr>
<tr>
<td>Design</td>
<td>17.4</td>
<td>18.7</td>
</tr>
<tr>
<td>Coding and debugging</td>
<td>22.1</td>
<td>21.7</td>
</tr>
<tr>
<td>System Testing</td>
<td></td>
<td>28.3</td>
</tr>
<tr>
<td>Documentation</td>
<td>12.2</td>
<td>11.3</td>
</tr>
<tr>
<td>Total effort</td>
<td>51.7</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Table II: Average percentages of effort by activity.

Figure 4 shows that the normalized percentages for COSMOS are comparable to those in industrial environments. If students were also required to perform the analysis and a system test, the relative allocation of their time would follow the 40-20-40 rule. Without these two activities,
however, their overall effort amounts to only half (51.7%) of the effort involved in a comparable industrial project.

To arrive at the productivity rate, we will divide the total number of statements in the finished product (exclusive development tools and test routines) by the total team effort in hours. We found that the average COS simulator consisted of 600 source statements independent of the choice of the implementation language (FORTRAN or PL/I). This simulator was completed after the third milestone and required an average effort of 157.8 hours (2 students times 78.9 hours). This corresponds to 3.8 source statements/hour. (The inclusion of 150 statements for the periphery test program in step 3 would raise the productivity to 4.75.) A complete version of COSMOS consisted of about 800 CAL statements. It was completed after the last milestone and consumed a team effort of 168 hours. The resulting productivity of 4.8 assembly language statements supports the hypothesis that the productivity in source statements/hour is independent of the type of programming language used. Overall, the finished simulation consisted of 1400 source statements and consumed 325.8 hours of team effort. Thus, a student's overall productivity was 4.3 source statements/hour.

Assuming language-independence of productivity rates, this overall rate exceeds the average industrial rate [2,10] by a factor 4.3. Since we found that students had to
accomplish only half of a comparable industrial effort (48.3% for analysis and system test did not apply), their normalized rate would correspond to 2.2 source statements per hour. The remaining factor of 2.2 may be attributed to a variety of aspects that are difficult to measure. The enthusiasm which the class generally displayed about the project surely affected productivity. The penalty for missing a milestone deadline and its effect on the course grade may also have had some effect. We would like to hypothesize, however, that the increased productivity is a consequence of the adopted design methodology. In essence, our analysis was not only concerned about the characteristics of the end-product, but simultaneously attempted to minimize the required implementation effort. Fortunately, these two goals do not require a compromise; the measures taken to reduce implementation effort actually improved the system architecture. In view of the resulting simplifications, the factor 2.2 is not surprising; it is generally known that less complex efforts boost productivity rates [2,4,10]. As an example, the designers of HYDRA attribute their high rate of 1.7 source statements/hour to their decomposition methodology [11] which aims at reducing the complexity of the implementation effort. A complex system is never simple to build, but the overall effort may be reduced by shifting the complexity to the analysis and design phases which are more manageable than the construction phase.
For completeness, we also give some statistics on the usage of computer resources. The final simulation ran in a region of 160 Kbytes of which the crossassembler took 50 Kbytes. The simulator itself occupied 70 Kbytes. The average student accounted for 50 hours of connect time, remotely submitted 100 batch jobs, and consumed 15 minutes of 370/168 CPU time. With 700 statements produced (excluding development tools and test programs) per student, one statement accounted for 4.3 minutes of connect time and 1.29 seconds of CPU time. Based on an effort of 162.9 hours per student, this also translates into 54 hours of connect time and 16.2 minutes of 370/168 time per man-month. One run (batch job) was needed for every 7 statements. Clearly, these resource statistics strongly depend on the specific project organization and computing environment. Industrial data vary considerably, partly because real-time systems often require testing in standalone mode. Wolverton [10] reports a need of 3 hours of 370/155 CPU time per man-month (the performance of a 370/168 is rated four times higher than that of a 370/155). Daly [2] found a need of 3 hours of preparation and analysis for each 1 hour spent testing on the machine in which the final product will reside. Together with his assumption that testing accounts for 22% of the development effort, this translates into 12.9 hours of CPU time per man-month (CPU time is assumed to cost from $40.00 to $150.00 per hour while the educational rate of our 370/168 is $1368.00 per hour). In view of the discrepancies
between the reported data, computing resource usage in software developments offers itself as a topic for further data collection and investigation.

CONCLUSION

The senior-level operating systems course at Rutgers University is coupled with an implementation project which involves both a hardware simulator and a multiprogramming system. In order to keep the implementation effort manageable, a major portion of the project analysis was concerned with the reduction of the implementation complexity. This task was aided by the concurrent design of the hardware and the operating system. This integrated design methodology emphasizes the choice of implementing lower level system functions in either software or hardware (or microcode, if applicable). While it complicates the analysis and lengthens the design phase, it simplifies the less manageable construction phase. This property suggests that the integrated design methodology may also be successfully applied to industrial projects.

Our statistics on the software development indicate that the distribution of effort over the various programming activities follows the 40-20-40 rule which is widely used to estimate industrial software development costs. However, since the analysis was performed by the instructor and only limited system integration was necessary, the overall effort for our instructional project amounted to only half of that
of a comparable industrial one. Because of this reduced effort, one would expect a student productivity rate of two source statements/hour. The difference to the actual rate of 4.3 statements/hour was attributed to the integrated design methodology.

Our statistics on computing resource usage are not easily related to published industrial data. In fact, these published data often vary considerably. In industry, computing costs are small compared to personnel costs. The converse is usually true in academic institutions where large class accounts may need to be justified. For this reason, we also presented data on the usage of computing resources. Statistics of other student projects will have to be investigated, however, before their general validity can be commented upon.
REFERENCES


