Saltzer

file - Multics design proposals

To: R. C. Daley

From: P. R. Bos

Date: February 29, 1972

Subject: Justification for the new online updater

Online installations involve changes to the system libraries while they are in use. This has the following implications:

(1) Segments which are to be removed from service may appear in the address space of existing processes, and hence may not be deleted, but must be preserved with all attributes except names intact until all such processes have terminated.

(2) The "window" during which a library is in an inconsistent state must be minimized. This means that if several segments are to be updated together, the name changes which effect the installation must be the last operations performed, the segments having all been previously copied, had acl's established, etc., rather than installing them one at a time.

(3) In the event of a system crash or other disaster during installation, enough information must be preserved in the directory to be able to repair the damage by hand. I.e. old names are to be transformed by some invertible operation such that the previous state of the library may be deduced by inspection.

(4) In the event an installation cannot be completed, the library must be restored to a consistent state as quickly as possible. I.e. error recovery must be complete and automatic.

(5) A detailed record of the installation should be kept, so that if for any reason error recovery fails, that record may be used as an aid in repairing the dmage (related to (3) above).

In the event that installed segments are unusable, it must be possible to undo the installation, subject to the same restrictions.

The current set of updating programs are deficient in the following respects:

(1) No provision exists for installing several segments as a single operation, so that there is a large window in

such an installation during which the library is inconsistent. This is evident even when replacing a single segment (an operation which involves two segments, the old and the new), because the previous segment is "deleted" before the new one is even copied; this can be fixed by rewriting the program, but the more general case cannot be.

(2) There is no provision for error recovery. This, together with (1) means that almost any error which causes an installation to be aborted results in the library being inconsistent. (Note that inconsistency appears not only between two object segments, but also between source and object for the same module.) This means that installations cannot be performed by absentee processes or by inexperienced personnel, due to the need for immediate manual error recovery.

(3) The current procedures for manual error recovery involve taking advantage of a bug in the way CACL ring brackets are handled (just as, previously, the installation process itself took advantage of the same bug). The alternative is to login ring_1_repair each time this happens, causing extra delays, or to do all installations in a ring 1 process.

(4) The current organization of source segments in >ldd makes large installations very time-consuming, and this must be prime shift time due to (2).

The only way at the moment of obtaining a cross-referenced (by bound segment name and component name) listing of the system libraries is via the MSL database and its associated software. This mechanism is deficient for the following reasons:

(1) The MSL has no provision for several entries having the same primary name component (e.g. an object segment and an include file). The primitive tools (lsm_, etc.) which manipulate the MSL segment are probably flexible enough to do this, but the msl_util interface is not.

(2) The MSL is rapidly approaching the 64K size limit; a number of include file cross-reference entries have already been deleted. Installation of pl1 version 2 will overflow the MSL.

(3) Due to system crashes and possibly bugs in msl_util or lsm_, an MSL is frequently made unusable, so that a previous copy must be retrieved. This results in loss of data, and by now there seems no hope of ever getting the MSL up-to-date.

The MSL can be eliminated by preserving more information (in particular, bound segment/component cross reference) in the directories themselves. Programs to replace "msl_info" (which interrogates a single MSL entry) and "msl_global_format" (which produces the printed MSL) using this information instead are currently being designed by Gary Dixon.

Large installations can be made less time consuming by reorganizing the source libraries such that source segments for a bound segment are archived together rather than alphabetically, so that the installation of a large bound segment requires only a single source archive update.

The remaining requirements have yielded the following design:

(1) An installation request may be broken down into a list of sub-tasks, i.e. installations of single bound components or unbound segments.

(2) The installation of a bound component may be broken down into a list of sub-tasks, i.e. compile the component, perform submission test on source and object, update (copies of) source and object archives, bind, perform submission test on bound segment, and install the source archive, object archive, bound segment, and bindmap.

(3) The installation of a single segment may be broken down into a list of sub-tasks, i.e. copy the segment (using a temporary name), set up the acl, remove names from the old segment (by renaming), add the new names to the new segment, and delete the temporary name from the new segment.

In each of steps (1), (2), and (3), it is desired to merge the lists of sub-tasks obtained at each level before processing any of them; this is done to satisfy implication (2) stated earlier. I.e. if we are to install several bound components, we will not install each one in turn, but will first do all the compilations, then all the submission tests, then update all the archives, then bind them all, and then install all the generated segments in one batch. Similarly, if we are to install several of these segments, they are not to be done one at a time, but first we will copy them all, then set up all the acl's, and only then will we touch any segments names in the library.

It becomes apparent, once the problem is stated in this way, that what is needed is a set of general-purpose programs which are capable of breaking down tasks into lists of sub-tasks, manipulating these lists, and processing them at some later time. Once having these, provided they are well-designed and easy to use, the updater requires only the following:

(1) A program to translate an installation request into a series of bound component updates

(2) A program to translate a bound component update request into a series of preparatory steps and a series of single segment updates

(3) A program to translate a single segment update request into a series of primitive tasks involving segment attributes, etc.

(4) Task primitives for the following functions:

- (a) list acl
- (b) add acl
- (c) delete acl
- (d) replace acl
- (e) list names
- (f) add names
- (g) delete names
- (h) transform names (name -> name.1 -> name.2 ...)
- (i) copy segment
- (j) update archive
- (k) bind
- (l) compile
- (m) submission test

In addition, (1), (2), and (3) are themselves task primitives at a higher level. As seen by the task list processor, all task primitives have the same interface, and so are interchangeable, and the task list processor can manipulate the interfaces without having to know anything about what is really going on.

Error recovery can be implemented trivially. For each task t, there exists another task t' which performs the inverse function (if only because that's the way we made them). Instead of generating a simple list of tasks, we can form a network, as follows:

$$\begin{tikzcd}
\text{listp} \arrow[r] & a \arrow[r] \arrow[d] & b \arrow[r] \arrow[d] & c \arrow[r] \arrow[d] & d \arrow[d] \\
& a' & b' \arrow[l] & c' \arrow[l] & d' \arrow[l]
\end{tikzcd}$$

If task a succeeds, we go on to task b. If not, we go to task a', and undo everything done by task a. If task b succeeds, we go on to task c; if not, we go to task b' and then to task a', again putting everything back the way it was. Since each task processed prior to the current task was successful, it is reasonable to expect that the inverse tasks will work also; hence this mechanism implements error recovery for an arbitrary lists of (invertible) tasks which should work in all but the most peculiar cases, and errors during installation are no longer a problem. Furthermore, it is possible to design the task processor and all individual task primitives such that processing may be continued (or error recovery invoked) after a system crash or process termination as long as the task list database is preserved (i.e. ESD works). In addition, de-installation can be implemented via the same mechanism used by error recovery.

For the initial implementation of the new online updater, it is planned to provide capabilities for installing groups of whole segments only, i.e. the topmost two layers of the tree will not be available at first. However, extending the initial implementation to the full mechanism described above involves writing only a new command interface and a few task primitives to update and bind archives and perform submission tests, and a task primitive to translate higher level installation requests into sequences of lower level tasks; the design is already complete in the structure provided by the task processor.

It is expected that the task processor (or another generation of it) may find wide application in other areas wherein tasks are broken down into subtasks which are not to be processed until later, or are to be processed in a different order than they are accumulated. Furthermore, arbitrary networks of tasks may be constructed, which can be thought of as "programs", whose instruction set consists of primitive tasks, in a higher-level language somewhat akin perhaps to an assembly-language version of lisp.

The schedule for completion of this project, to be perfectly honest, is very uncertain, since all components of the updater are dependent on the same task list processor. Once that is completed, things should progress well, but because the task processor must be the common denominator between all other modules of the updater, without itself becoming hopelessly fragmented, obtaining a workable design has been a major problem. At the present time, I believe that the last major design bug has been resolved, so that final code for the task processor (totaling perhaps as much as 150 or 200 executable pl1 statements) may be completed. The name and acl task primitives were coded and partially debugged for a previous incarnation of the task processor, and should require only minor changes to interface to the new version. Also, the algorithm for breaking down a segment installation request into these primitive tasks is known, although it is not at present coded in any usable form. Finally, the argument processor of the command interface has been almost completely coded, but the design of the rest of the command must wait for all lower-level task modules to be completed.

If all goes well, it is possible to complete the updater on schedule. However, the schedule is optimistic; unforseeable problems may yet crop up, as happened just a few days ago. On the whole, this project has become about five time larger than first anticipated; online installations are not as trivial as some management personnel seem to feel. Nevertheless, the current installation programs are totally unsuited to the task, requiring a very high degree of proficiency on the part of the installer to cope with all the things that can go wrong (what will we do if Arlene has to leave?). For this reason, I feel that though this is not by any means a highest-priority task, it cannot be put off for very long, and work should continue even though it is taking a long time.