801 Storage: Architecture and Programming

Albert Chang and Mark F. Mergen
IBM T. J. Watson Research Center, Yorktown Heights, New York 10598

Extended Abstract

Based on novel architecture, the 801 minicomputer project has developed a low-level storage manager which can significantly simplify storage programming in subsystems and applications. The storage manager embodies three ideas: (1) *large virtual storage*, to contain all temporary data and permanent files for the active programs, (2) the innovation of *database storage*, which has implicit properties of access serialization and atomic update, similar to those of database transaction systems, and (3) access to all storage, including files, by the usual operations and types of a high-level *programming language*.

Multics first suggested that files be placed in virtual storage. Programming is simpler because all data may be directly addressed by the processor and because the operating system performs all disk I/O in response to page faults. Processes share one copy of files in a system-wide real storage buffer.

The transaction concept, a sequence of data access or update actions grouped as a unit, comes from database systems, e.g. IMS and System R. If many transactions share data and run concurrently, systems often guarantee that the results are *serializable*, *atomic*, and *permanent*, i.e. as if transactions run one after another and, after failure, each is complete or has no effect. This simplifies recovery and helps maintain database consistency. We add the idea of *fine granularity protection hardware* which invokes typical transaction software mechanisms of locking and change logging. Our approach combines virtual access to files with implicit transaction functions.

The IBM RT PC implements the necessary features of 801 storage architecture. The upper 4 bits of a 32-bit *short address* select one of 16 segment registers. A 12-bit *segment id* from the register replaces the 4 bits to form a 40-bit *long virtual address*. This creates a single large space of 4096 256M-byte segments. Only the supervisor may load segment registers and may therefore control access to and sharing of segments. A long virtual address is translated to real by an *inverted page table*, in which each entry contains the virtual page address currently allocated to a real page. Hardware searches the table using chained hashing. If a given virtual address is found in a table entry, the index of that entry is the desired real page address. Table size is related only to real storage size rather than to virtual size as with conventional segment and page tables. The inverted page table includes a *transaction locking mechanism*. Each entry contains bits to represent read and write locks, for *128-byte lines* within the page, granted to the *transaction id* also in the entry. Lock fault interrupt occurs when storage access by the current transaction (id in a register) is not permitted by locks and id in a table entry. Page protection bits may instead be used when transaction locking is not required.

CPR is a prototype operating system for research in several areas, with major emphasis on storage facilities, which was developed on prototype hardware and now runs on the RT PC. The storage of a CPR process is a subset of segments in the single large address space. Process state includes the segment registers but, by loading segment ids, a process may access many more than 16 segments.

The *working storage* of each process (active programs and temporary data) is contained in up to 8 segments whose ids are always loaded. This allows all access to working storage, including linkage and parameter passing, to use only 32-bit *short addresses*. Two segments, which contain the supervisor, extensions, and read-only parts of commonly-used programs, are shared by all processes. Other working storage segments are private to each process. CPR working storage is similar to that of widely-used systems like DEC VMS and IBM MVS.

Many programs and temporary variables may occupy the same segment. By contrast, when *files* are opened, they are each *mapped* into a separate 256M-byte segment. Processes who open the same file share the segment id. *Open* does not return the segment id but instead returns a capability to it called a *refp*. The supervisor loads a file segment id when requested, after verifying the *refp*. Since file segment ids are not always loaded, a file storage address is a *long address* of the form [*refp*, offset in segment]. Because of this indirect reference to segment ids, if the hardware were enlarged to provide more total segments, only a few supervisor programs would need change. Up to 6 segment registers are used by a process for its many open files. Page faults in file storage cause the storage manager to read pages from the disk copy of the file. CPR file storage is similar to that of Multics.

A CPR transaction is all the storage actions by a process on a set of file segments, performed between two calls to *commit* for that set of segments, including actions of the supervisor in directory segments when files are created, renamed, etc. File *open options* influence transaction processing. When open options specify file sharing and implicit functions of serialization and atomic commit, we call the file segments *database storage*. Transaction locking hardware is used. Lock fault interrupts invoke the storage manager to grant locks and later, when a transaction commits, the storage manager writes log records of changed storage. Strong transaction properties are achieved, without explicit calls from programs which access storage and independently of superimposed data organization or access pattern. A language-based approach to invoking transaction functions seems to be more explicit and restrictive.

PL.8 is a PL/I dialect for systems programming and the PL.8 compiler is part of the 801 project. To simplify file programming, a *persistent* storage class (i.e. storage in a file segment) and a *refp* type (capability to a file segment id) were added to PL.8. Persistent storage may be used with any type, including arrays of structures and based structures in areas (appropriate for records, indices, etc.). Semantics of computation with persistent variables depend only on type and so are the same as computation in working storage classes automatic, static, and controlled (heap). Computation with large aggregates over multiple file segments is possible, exceeding 32-bit addressing. The language hides most distinctions between short and long addressing and the compiler manages addressing details, including loading of segment registers. Called procedures are independent of any transaction functions required in variables passed to them, because transaction locking is invoked by the hardware assist. Language extensions are not required to address CPR file storage. *Refps* may be declared with existing types, segment registers may be explicitly managed, and storage may be addressed with 32-bit pointers in Pascal, C, assembler, etc.

Our prototype experience is favorable. The RT PC storage control chip implements the architecture, with less than 10% of area for transaction locking. The CPR storage manager is less than 15K lines of PL.8. No-conflict lock grant requires 250 instructions. Load segment register with verification requires 25. A version of the debit-credit transaction benchmark was implemented in PL.8 with file sizes appropriate to our hardware. Instructions and disk I/Os per transaction are comparable to fast and good categories reported in the literature. Attanasio implemented SQL/801, a large subset of the SQL relational database language (most verbs, n-way joins, etc.), in less than 15K lines of PL.8. The program is written as if there were only a single user but achieves multi-user concurrency by placing tables and indices in database storage. The performance of SQL/801, with the DeWitt database benchmarks, is approximately equal to that of SQL/RT, a conventional product implementation on the AIX operating system. The hardware and software locking cost when SQL/801 runs these benchmarks is less than 5% of elapsed time. More work is needed to explore properly the limits, e.g. concurrency issues, and parameters of performance.

Database storage is a new way to implement certain storage management functions in an operating system, built on and similar in spirit to virtual storage. Both are very general, transparent, and rather monolithic approaches to storage management, one for storage hierarchy, the other for storage concurrency and recovery. We believe that database storage will perform well for a wide range of applications and that the simplicity it offers is too attractive to dismiss. As in the early days of virtual storage, the challenge is to understand and exploit its characteristics.