An Overview of MAGIC 6

MAGIC 6 is the current name of the operating system in development at the Architecture Machine for use on the thirty two bit Interdata machines. The system provides a multitasking environment which is suited for the execution of PL/I (and PL/I compatible) programs. The system provides memory management, execution management, dynamic linking, device management, and on-line storage management. The system requires a 7/32 with the memory access controller.

The basic unit of the system is the task. A task consists of a saved machine state and an address space stack. An address space consists of up to sixteen segments, each of which may be up to 64K bytes in length. When a task is running, its procedures may access any data in the segments in the current address space. Since sixteen segments is a rather small number, the address space may be "pushed", which frees up a specified set of the segments in the current address space. When the address space is "popped", the previous address space becomes the current one.

Each task may have a different access to a given segment. The accesses are read, write, execute, and gate. In reality the access control is a bit more sophisticated, but these four are sufficient for this discussion. The first three acess modes are rather straight forward. Gate access is used to allow the system to be entered safely. A segment with gate access can not be read or written. If it is branched into a fault will occur, and the address branched to will be "validated" to make sure that it is at a valid entry point, before resuming execution in the domain associated with the gate segment. Currently, this gate feature is only being used to protect the system, but may be expanded in the future.

When a task is created, it is initialized with a certain number of segments (unless this is explicitly over-ridden). These include the system itself (in a segment called "HCS"), the process data segment which contains the stack for system program execution among other things, the users stack, and the users linkage segment. In addition, the PL/I operators are also initiated in the created segment. This leaves eleven segments for the users programs. The user's stack is a segment which contains the PL/I stack for the user's programs. The user's linkage segment contains all per-process, per-procedure storage, such as static internal storage. It also contains the impure code fragments needed for dynamic linking.

Dynamic linking is supported by both the PL/I compiler and the macro assembler. Basically, references to external routines are turned into calls into the linkage section, which are initialized to contain "unsnapped links" which are faults to the supervisor (which use the SVC mechanism). When a fault is executed, the link to an entry of the form segment$entry is broken into a segment name and an entry name. The segment is initiated (if not already so) and the link turned into a branch to the appropriate entry. Thus, the bulk of the code can remain pure while still supporting dynamic linking without indirection.

This dynamic linking feature allows the system to consist of a number of user callable routines to initiate, terminate, and modify segments as well as to perform the complete repetoire of system functions. In general, the system will reside in one particular segment. Currently, external variables are not supported. In the future, they will be handled by the dynamic linker with a few modifications of the PL/I code generator.

When the dynamic linker wants to locate a segment in order to snap a link, it first checks to see if that segment is already known to the task. If it is not, then that segment is searched for in a series of directories called the task's search rules. The object file in question is then initiated. An initiated segment need not remain in core at all times. In reality, segments can be pushed out of core and written back into the file system if they have been modified. If they have not been modified since last loaded, then it is possible to simply reuse their core storage and to simply reread them upon the next reference.

The core shuffler task is responsible for handling any memory allocation beyond the simplest growth of a segment. In a similar fashion, the disk servor task is responsible for disk shuffling and the queuing of requests to the disk drive. The system initiator task is the global parent task, of which all other tasks are subtasks. This task is created upon the system coming up, and establishes the disk server and core shuffler tasks before any users can use the system.

The file system itself is based on 2K byte blocks. The file manager and directory manager are the two main levels of the file system. The file manager consists of routines to manage the VTOC (Volume Table Of Contents) which contains entries which in turn describe the files. The VTOC itself is simply a special file as is the allocation mask and the root directory. Wheaever a file is created in the VTOC it is assigned a unique identifier which concatenated with the index of the entry in the VTOC forms the file id. This file id can then be used to refer unambiguously to the file in question. Thus, to associate an active segment with a file on the disk, all that is needed is file id for that file (and the file id of its containing directory and the files directory entries offset so that access changes and the like can be effected through the file system).

The directory manager uses the file file manager to allow names to be associated with particular files. Currently, directories are limited to 64K bytes in size so that they can be initiated as segments by directory management routines. The directories are basically PL/I areas with special headers so that the hash table and the like can be located and used to convert the name of a file into saved information about it and its file id.

To aid in recovering from disk crashes, the file system will also maintain a cylinder descriptor on each cylinder which will describe its contents and the most recent times its blocks were written. In particular the file id and the block number of each block written to a cylinder will be stored in one block on each cylinder.

Other devices such as tape drives, IMLACs, tablets and the like will be controlled by low level system routines which can be dynamically linked to. The user oriented I/O system will be stream oriented, allowed data to be transmitted either as ascii or binary data either to devices, or through manipulating programs. The graphics system will be incorporated in this scheme by also allowing streams to have graphically oriented data to be transmitted across them. Streams will have names which will correspond to PL/I files so that PL/I I/O can be used by those familiar with it. Naturally ioa and ask will be around.

The central "file processor" will handle a number of remote user processors which do not have file systems. The system running in the remote machine depends on its architecture. The sixteen bit machines will be running a modified MAGIC 4, which will provide a high level interface to the new file system. Memory management, local device I/O and the like will be more or less unchanged from their current states. With a few exceptions, PL/I programs which currently run in sixteen bit mode will continue to do so.

The remote machines will be communicated with via the existing interprocessor bus. Each remote machine will have a task running in the central machine which will act as its liason to the file system itself. This task will initially be general purpose, but if it would improve efficiency, it could try to "out-guess" the remote processor, and have started work on the next step before it is needed.

As far as program preparation goes, certain tasks are best done in the remote machines, and others done in the central processor. There is no reason to ever transmit any data to the remote processor in order to copy one file into another. Tasks, such as editting, which involve high interaction rates could be done in the remote processors. This can almost be extended to IMLACs which are actually rather nice editting machines although we are currently limited by software. Thus, if an IMLAC were hooked up to the central processor, it could be used for editting files with almost no load on the central processor.

Tasks such as compilations and assemblies are usually I/O and core bound rather than CPU bound. Since we will have compilers which can run in either the remote or central processor it may be advantageous to experiment in order to find the best place to run them. In theory, two people doing PL/I compilations could benefit by running on the central machine since the compiler would only have to be moved from disk to core, and not twice again to core. If the central processor has sufficient memory, programs such as the PL/I compiler and the like would stay in core for sufficiently long periods of time which would improve performance with either a local or remote compiler.

Thirty two bit remote processor users will be running a version of MAGIC 6 with a file system write around which makes the actual attachment of the disk drives to the machine transparent. Since certain routines will be used frequently both in the remote and central machine references to them should rarely require disk references. Even MAGIC 4 has a small enough working set to allow 64K bytes of disk cache to cut disk references by 50%.

At this time it is impossible to give precise numbers for the data transfer rates and number of remote users to saturate MAGIC 6. Too much depends on the configuration. If the central machine has an extra selch, or is an 8/32, or if we dedicate one machine to compiling and linking PL/I programs all of the numbers will change dramatically. The current accepted figure for human interaction is that a given user only uses the computer 20% of the time spent sitting in front of it doing anything. Observation reveals that the current debugging loop ranges from a few minutes for a small set of programs to fifteen to twenty minutes for systems such as PAINT or MAS. Once the program is in core and running it places few demands on the file system. People either stare at the edittor, plod, or the program itself.

One of the main concerns of the new system involves minimizing the "turn around" time required for making changes in programs. Dynamic linking helps here, although the limited number of segments makes it a little more difficult to use than on Multics. (Most big systems here have at least 20-30 modules as opposed to about ten user usable segments.) On the other hand, many gains can be made by distributing the labor better. As Mike Miller suggested, most of the YONA system could be run in the IMLAC itself, if we had the software so that a relatively naive programmer could learn how to program it. Similar problems come up involving the use of the 85 and the "super-graphics" machine.

It is dangerous to speculate too far or too deeply into the future without sufficient knowledge. This September, MAGIC 6 will be more or less running in a nuclear form. Once an editor (TINT), a compiler, and an assembler are running on the system, it can be rounded out under its own power. It will probably not be generally usable until January, and even then, the graphics system must be designed, dozens of little systems programs must be written, and the system itself must be shaken down before any but the hardiest users should rely on it. Talks with various people working on various projects have helped us get so far. Ideas on a new graphics system, a graphical programming language if any, library routines, improvements in memory management in remote machines and the like are welcome.