Burroughs

BSP

CONTROL PROGRAM

BSP

BURROUGHS SCIENTIFIC PROCESSOR

CONTROL PROGRAM

CONTENTS

ABSTRACT

The control program for the Burroughs Scientific Processor exploits functional distribution to provide a full-featured batch and time-sharing operating system for general-purpose and interaction-intensive computing, while providing an efficient monoprogramming environment for high-speed, parallel, numerical computation. Efficiency is obtained by incorporating physical I/O functions into the I/O controller and performing most scheduling functions on the separate system manager. The inefficiency of demand-paged virtual memory is avoided while retaining its user convenience by providing static memory allocation and overlay coupled with a sophisticated linker that estimates working sets via source program analysis. Finally, the scientific environment is integrated into a general purpose system via Burroughs high-level job control language and multiclass priority scheduling system.

1. BACKGROUND

The development of the Master Control Program (MCP) for the Burroughs B 5000 in 1961 marked the beginning of the present generation of operating systems, providing as it did full support for multiprocessing, virtual memory, time-sharing, and a host of other system and user conveniences that have since become commonplace throughout the industry. But even as the mainstream of computing moves in the direction of the multiuser, short response time environment, there remains a growing backlog of scientific applications for which the fundamental throughput limitation, now and for the foreseeable future, is raw arithmetic speed. This latter environment is the domain of the supercomputer – the high-speed number-cruncher whose productivity is measured simply as the number of floating-point arithmetic operations performed per unit time. For such applications, even a supercomputer strains to support even one program at a time, and the typical general-purpose operating system, with its concommitant overhead, is as likely to reduce system throughput as to enhance it.

Against this background, the Burroughs Scientific Processor (BSP) debuts as Burroughs first commercial entry into the supercomputer marketplace. But the BSP is not Burroughs first parallel supercomputer. Burroughs engineered and constructed the ILLIAC IV – one of the fastest machines ever delivered – and more recently built the PEPE system hardware under subcontract to Systems Development Corporation.

The BSP is not merely a supercomputer. Integrated as it is with a Burroughs conventional, large-scale system as a host, the BSP system represents a total-system solution and offers an exceptional arithmetic capability within a fully-featured, general-purpose computing environment.

2. FUNCTIONAL REQUIREMENTS

The functional requirements for BSP system control software can best be understood in terms of the typical scientific supercomputer workload.

First, supercomputer programs are long-running. Supercomputers are typically dedicated to a few production codes that would run for many hours on present-generation large-scale computers. This contrasts with large numbers of short jobs characteristic of a general-purpose scientific or engineering environment. Thus, the environment dictates that a supercomputer achieve maximum performance on a single program, rather than just some average throughput over a mix of programs.

Second, supercomputer programs require large amounts of main memory but only moderate amounts of secondary memory. A typical data base is a large array representing a large matrix or a grid of points in physical space. Data base sizes may reach millions or tens of millions of words, but rarely require hundreds of millions. Data are usually accessed in a regular pattern, with a complete computation requiring many passes of the array. On a single pass, however, the instantaneous "working set" of references may span a large cross-section of the array, with only a few references per individual datum. Thus, it may be desirable to contain the entire array in main memory; alternatively, cross-sections of the array can be streamed through the main memory from secondary storage. This contrasts with many data processing applications which have very large, randomly accessed data bases, but require relatively little main memory to process a single transaction. Thus, the environment dictates large main memory capacities, backed up by a secondary storage about an order of magnitude larger with a very high, effective, sequential transfer rate.

The BSP system software, no less than the hardware, contributes to and benefits from this total system approach. The host system Master Control Program is as comprehensive in function as any operating system in the industry, being the culmination of nearly two decades of operating system development. Coupled with the MCP is the system software for the BSP proper, whose main objective is to convert the hardware number-crunching payload – expressed in tens of floating-point operations per microsecond – into a proportional system throughput measurable in trillions of arithmetic operations per day.

The main system software contribution to computational throughput is, paradoxically, minimizing the software presence.

Traditional sources of macroscopic delays to programs have been input-output operations and state-switching arising from multiuser, interactive environments. The input-output delays are minimized by a hardware approach that combines charge-coupled device (CCD) technology, which provides superior device performance characteristics, with a sophisticated file controller that totally eliminates operating system software overhead for routine block transfer operations. State-switching delays are minimized by providing a separate environment – the general-purpose host processor – for multiuser, interaction intensive activities such as program preparation, editing, and compiling, thus allowing arithmetic-intensive production programs undisturbed exploitation of the high-speed computational resources of the BSP proper.

BSP

BURROUGHS SCIENTIFIC PROCESSOR

FAULT-TOLERANT FEATURES

3. FUNCTIONAL CHARACTERISTICS

Based on the workload considerations discussed in section 2, the basic tenet of the BSP design is to provide continuous execution of suitable scientific programs at the rated processor speed. To this end, the file memory (FM) device provides access time and transfer rates sufficient to sustain fully overlapped sequential I/O at a computation-to-I/O ratio as low as 10:1, and provides capacity to contain all files required by a typical program on this high-performance device. The goal of BSP system software is to support this hardware approach for scientific programs while integrating it into the overall philosophy of the Burroughs large-scale system.

COMPUTATIONAL ENVELOPE

A characteristic feature of the BSP approach to fast and efficient scientific computation is the concept of the computational envelope. By this, we mean that a scientific task, once started, runs to completion within the high-performance computational and I/O environment of the BSP without requiring intervention of or access to the much slower system manager processor or its I/O devices. In particular, all BSP FORTRAN program and data files are normally fully contained within file memory while the program is in operation; input and output files are copied to or from file memory before the task is started or after it completes, respectively. (Means are provided, however, to access exceptionally large files which cannot be fully contained on file memory.)

FUNCTIONAL DISTRIBUTION

A primary requirement for system software on a number-cruncher, particularly a parallel computer on which control software runs at relatively slow scalar speeds, is simply to stay out of the way. Consequently, many system functions contributing to software overhead on conventional systems have been removed from the computational envelope or incorporated into hardware for asynchronous execution. For example, several operating system functions are off-loaded to the system manager, including low-speed peripheral spooling, permanent file catalog management, file copy between file memory and the system manager, work-flow (job control language) interpretation and much of job scheduling and operator console support.

Control program overhead for block-level I/O operations (read and write) has been completely eliminated by performing the necessary functions in the file memory controller, including priority request queueing, access rights verification, logical-to-physical address translation, error retry, and completion notification.

MONOPROGRAMMING

Since the BSP is designed for continuous running of single programs, the BSP system software is likewise oriented to efficient processing of one program at a time, as opposed to reliance upon multiprogramming to achieve high throughput. Nevertheless, the need for smooth transition between tasks and priority service to urgent tasks is recognized and supported. Accordingly a number of BSP programs, with their data files, may be queued on file memory; they will be run to completion in priority sequence. Furthermore, particularly urgent programs, such as short debug runs, may preempt a running task.

PLANNED OVERLAY

The BSP hardware does not provide virtual memory. Accordingly, programs too large to be memory-contained must be statically divided into overlays. The BSP linker supports this capability by analyzing program control flow and dividing the program into overlays consistent with a specified maximum memory size. Linker-generated modifications to the subroutine-calling sequence invoke the overlay supervisor, so no special providions for overlay are required in the FORTRAN source program.

INTEGRATED JOB FLOW

Although the BSP and its system manager function internally as independent asynchronous processors, a user views the total system as an integrated whole. There is a single, job control language – Burroughs Work Flow Language (WFL) – for describing job sequences involving both the system manager and the BSP, into which BSP functions have been integrated in a natural and consistent way. A single operator console manager permits the entire system and its scheduling policies to be monitored or controlled from any operator console terminal.

4. JOB FLOW

Figure 1 illustrates a typical job flow. (Numbers in the text are keyed to the figures.) A typical session begins with a user logging on at a remote terminal connected to the system manager's data communication network (1). The user then interactively creates new programs and data files, or modified previously stored files, using the Burroughs large systems Command and Edit Language (CANDE) (2). (Programs can also be submitted via a local card reader (3)). When all program and data file updating is complete, the user initiates the necessary sequence of compilation, file copy, and program execution tasks to carry out his particular computation by starting a job. A job is itself a program which the user would have previously written using the Burroughs Work Flow Language (4), which invokes various system and user programs in specified sequence and defines the system resources and files required for each. Starting a job causes the WFL program to be compiled into executable code (5). Once the job is started, the user continues with other work while awaiting completion and results.

In the meantime, the system manager Master Control Program (MCP) queues the job (6) and, based on the job's priority and resource requirements, begins its execution (7).

A typical job begins with a BSP FORTRAN compilation (8) (or several compilations running concurrently), followed by execution of the BSP linker (9) to bind these and previously compiled subroutines into a single executable program. Concurrently, input data files for the BSP program are copied to the BSP file memory from permanent storage on the system manager (10). All of these activities take place on the system manager, while other work unrelated to the job is in progress on the BSP.

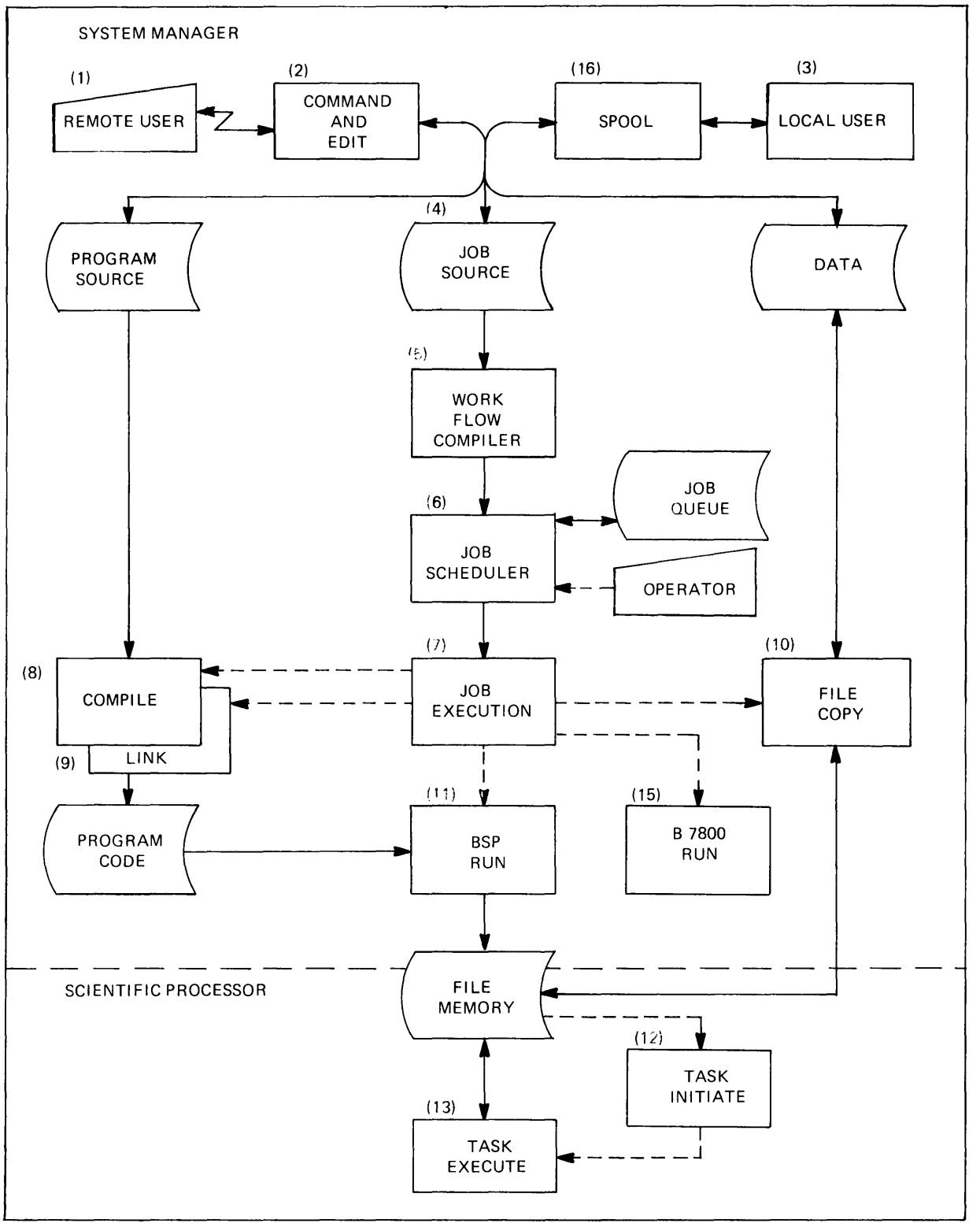

Figure 1. Typical BSP Job Flow

When these activities finish, the system manager, responding to a single Work Flow Language statement, initiates the BSP program (11). The program and card-image input files are automatically copied to file memory, and the BSP task is queued by priority for execution on the BSP (12). The task begins as soon as all higher priority BSP tasks have completed (immediately if it has preemptive priority over a lower priority running task) and runs to completion (unless pre-empted) (13). When the task completes, the BSP sends timing and performance information to the system manager. The system manager MCP automatically copies printer and punch spool files from file memory to the system manager, and resumes execution of the job.

The next tasks in a typical job will copy permanent output files from the file memory to system manager permanent file storage (14). Finally, data editing or analysis routines may be performed on the system manager (15). When the job completes, the system manager MCP will automatically spool bulk hard copy output to appropriate peripherals, and print a job summary listing giving job accounting information, resource utilization, and performance statistics (16).

Figure 2 illustrates the organization of the MCP functions invoked via the job flow process. The MCP consists of four major components: the Work Flow Language compiler (7), the controller which performs scheduling and console management (2), the system manager MCP proper (3), and the BSP MCP (4). (The first three of these components run on the system manager.) The BSP MCP implements two major functions: task scheduling and loading (5), and file memory management (6). The I/O subsystem functions (7) logically form a third function of the BSP MCP, but are physically realized largely by hardware.

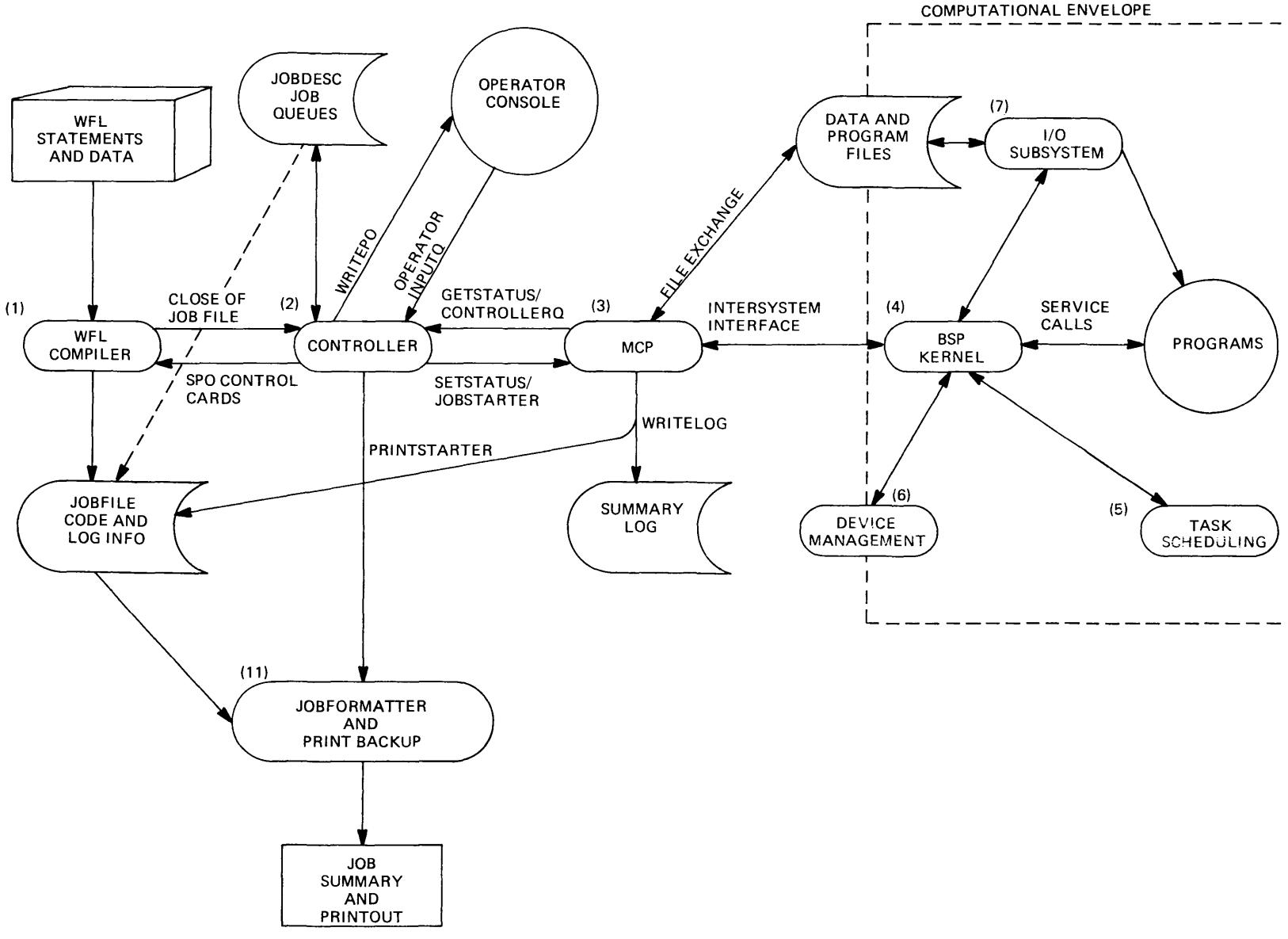

Figure 2. Job Flow, System Viewpoint

5. WORK FLOW LANGUAGE

The Burroughs Work Flow Language (WFL) is a high level, ALGOL-like language for describing the particular sequence of program executions that constitute a job. A partial list of WFL statements and their associated functions is given in Table 1.

Table 1. Representative Work Flow Language Commands

Statement	
JOB	Identify a job
USER	Provide accounting and security identification
CLASS	Classify a job for default resource assignment and scheduling
BEGIN... END JOB	Delimit a job
SUBROUTINE... BEGIN... END	Identify and delimit a subroutine
IF... THEN... ELSE	Conditional construct
WHILE... DO	Looping construct
RUN	Initiate a synchronous program execution

Table 1. (Cont'd)

Statement	
PROCESS WAIT	Initiate an asynchronous program execution wait for asynchronous program completion
COMPILE ... AND GO	Initiate a standard compiler, optionally followed by execution of the compiled program
COPY	Initiate the standard file copy utility
DATA... ?	Delimit an in-line data file
RESTART, SPRESTART	Designate a restart point for resumption after system failure
FILE	Identify files used by a program
CORE, PRIORITY, SPTIME, JOB RESOURCE, etc.	Job or program resource requirements (See also Table 2.)

Figure 3 illustrates a typical Work Flow Language program for a simple "compile and go" run. In this example, the CLASS statement indicates that this job will operate with default resources (file memory, processor time, priority) previously established by the installation for that class of job. The input data is included with the job source, and the printer output will be handled automatically. Thus, for this simple situation, only a minimum number of work flow statements is required.

Figure 4 illustrates the Work Flow Language necessary to execute a more complex job such as the one discussed earlier in section 4. This illustrates the power of the Work Flow Language to specify resource requirements, compilation and linking sequences, data file copy operations, and error restart points.

```
?   JOB                 JOB/NAME:
    USER                USERCODE/PASSWORD;
    CLASS 1;
    COMPILE             SAMPLE/PROGRAM VFORTRAN AND GO;
    FMDATA;             % (delimit a data file for file memory)*
      .
      .                 MISC DATA FOR PROGRAM
      .
?   END JOB
```

*Text following % is commentary, not part of the Work Flow Language

Figure 3. Typical Work Flow Language Program, Simple "Compile and Go" Run

```
1000        ?.  JOB SAMPLE/PROGRAMS
1100            USER USERCODE/PASSWORD;
1200            CORE = 25000; (% system manager memory requirement)
1300            JOBRESOURCE (FILEMEM=2000000);
1400            PRIORITY = 80
1500            BEGIN
1600                COMPILE SUB/PHYSICS VFORTRAN TO LIBRARY;
1700                    VFORTRAN FILE TAPE = SOURCE/PHYSICS ON OURPACK;
1800                    DATA CARD;
1900                    .
2000                    .    SOURCE PROGRAM PATCHES
2100                    .
2200        ?
2300                LINK WEATHER/CODE LINKER;
2400                    DATA CARD;
2500                    $    LINK SUB/PHYSICS
2600                    .
2700                    .    OTHER LINKER CONTROL CARD
2800                    .
2900        ?
3000            ON SPRESTART GO COPYFILES;
3100            COPYFILES:
3200                COPY WEATHER/CODE TO FILEMEM;
3300                COPY INPUT/ATMOSPHERE FROM OURPACK(PACK) TO FILEMEM;
3400                RUN WEATHER/CODE;
3500                    SPTIME = 30;
3600                    FILE FILE10 = INPUT/ATMOSPHERE ON FILEMEM;
3700                    FILE FILE11 = OUTPUT/ATMOSPHERE ON FILEMEM;
3800                    FMDATA FILES;
3900                    .
4000                    .    MISC DATA FOR PROGRAM
4100                    .
4200        ?
4300                COPY OUTPUT/ATMOSPHERE FROM FILEMEM TO OURPACK(PACK);
4400        ?   END JOB
```

Figure 4. Typical Work Flow Language Program, Complex Job

6. SCHEDULING AND OPERATOR CONTROL

The system manager control program has been designed to provide fully automatic system scheduling, including the BSP, within policies established by the installation manager. On the other hand, status monitoring and operator intervention can be maintained at multiple locations via local or remote operator display terminals, each individually programmed to provide the desired status information and to accept selected commands.

System scheduling is implemented via a multiclass priority queue mechanism, coupled to automatic spooling for collecting jobs from or disseminating them to local peripherals or remote terminals. By this mechanism, the installation manager may designate various classes of jobs, each with particular default and maximum resource requirements. For example, the installation could designate a high priority class for short, small jobs and a lower priority class for large or lengthy jobs. The mechanism also allows an installation to collect and queue certain classes of jobs during the day for later execution at night. Table 2 lists selected criteria that may be used to establish job classes.

BSP tasks in particular are normally initiated in priority sequence and then run to completion. However, the installation may designate a certain class of job which may preempt a lower-priority BSP task, causing it to be automatically rolled back to file memory and resumed when the preempting job finishes. This mechanism provides fast turnaround for urgent work while preserving the basic efficiency of balanced monoprogramming environment.

Although the system schedule makes routine operator scheduling decisions unnecessary, the operator can override the scheduler in exceptional circumstances. A full repertoire of status displays and control commands is available.

Table 2. Typical Job Scheduling Attributes

Attribute	Description
PRIORITY	Priority for processing time
PROCESSTIME	Processor execution time limit
IO TIME	Input-output time limit
ELAPSEDLIMIT	Elapsed "wall-clock" time limit
LINES	Lines printed limit
CARDS	Cards punched limit
DISKLIMIT	Disk space utilization limit
JOBRESOURCE	File memory utilization limit
SPTIME	BSP execution time limit

7. FILE MEMORY ALLOCATION

The file memory (FM) is one of the BSP's most precious resources. Since very large capacities are uneconomical with the present state-of-the-art, particular effort was invested to provide efficient space allocation so the file memory could serve as both file storage for an active program and a staging area for previous or future tasks.

The allocation algorithm selected is a modification of the Banker's algorithm made famous by Edsgar Dijkstra, now a Burroughs Research Fellow. The objective of this policy is to make the file memory available to as many requestors as possible without exposing the system to over-commitment, thus requiring a costly rollback of some file in order to make enough file memory available for a running program to complete. This is possible if each job, before making any requests, informs the allocator of the maximum FM usage for that job; the Work Flow Language provides for this limit. The Banker's algorithm, then, can allow files for several jobs to be staged to file memory, reserving only enough additional space to satisfy the working storage requirements of one job at a time, yet guaranteeing that each job in turn will have sufficient working storage.

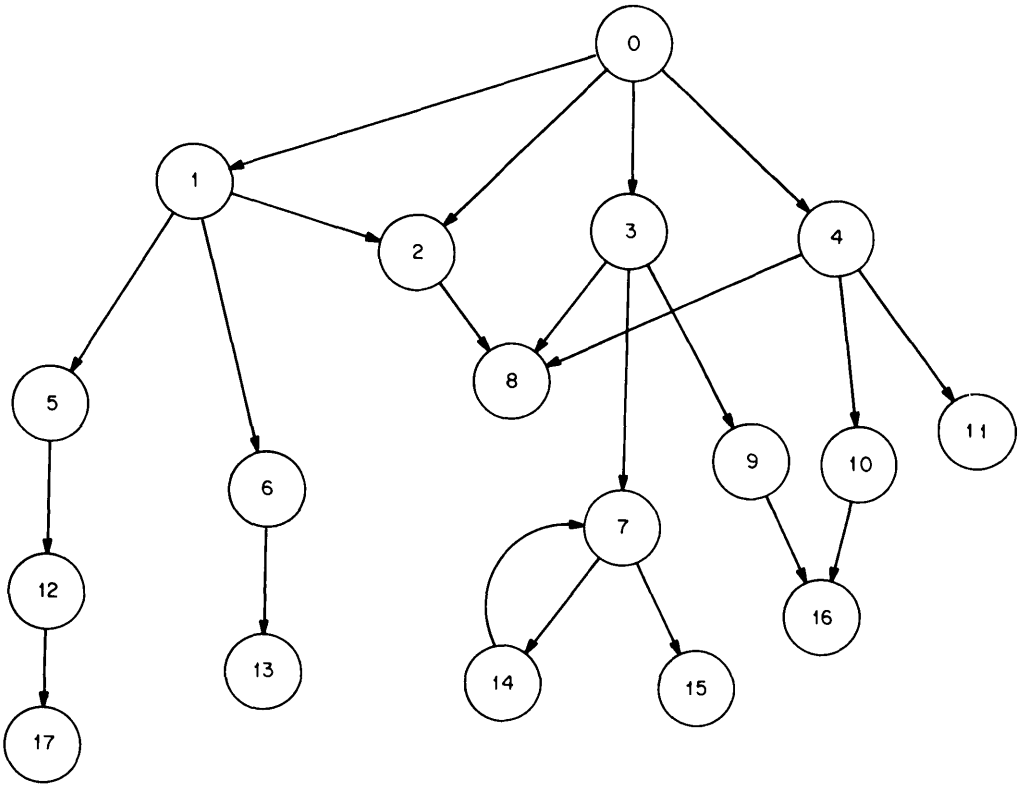

Figure 5. Typical Program Structure for Overlay

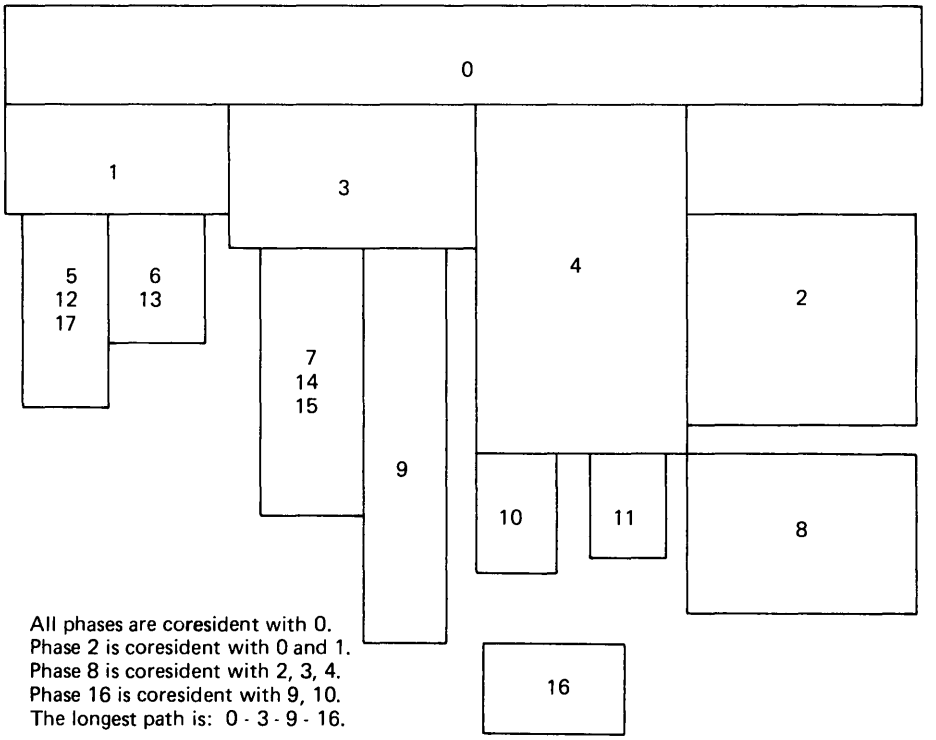

Figure 6. Typical Overlay Memory Map

8. PROCESSOR MEMORY ALLOCATION

The philosophy of processor memory allocation is to minimize interference with scientific computation due to memory unavailability. The hardware provides for exceptionally large memory capacities — up to 8 million words. Memory interference is minimized by providing separate array data and program instruction memories. The monoprogramming philosophy makes the maximum memory available to an individual program. Nevertheless, software must provide for occasions when the total size of a program exceeds processor memory space.

The memory overlay scheme for the BSP has been designed to provide most of the user convenience of a virtual memory environment, while providing the efficiency of a planned overlay environment. To this end, a separate software component, called the linker, is provided. The linker binds one or more compiled FORTRAN subprograms with referenced library routines into a single load module. In addition, the linker analyzes the cross-references to subroutines, common blocks and I/O units to determine which blocks need be coresident and which can overlay each other. (Figure 5 illustrates a typical program graph.) Given a maximum memory availability, the linker will then partition the load module into overlay phases. (Figure 6 illustrates the overlay phases for the structure represented in Figure 5.) The linker also inserts calls to the overlay supervisor preceding initial interphase references. Thus, a static overlay structure is produced with no user modification whatever to the FORTRAN source program. If the user wishes to override the linker's algorithm, he may specify an explicit overlay structure via control cards to the linker; this still requires no change to the FORTRAN source.

The memory allocation patterns provided by the linker include provision for automatic allocation of local variable space, including dynamically specified array dimensions, and provision for SAVE memory, so values computed by one invocation can be retained for the next invocation. The linker also exploits hardware

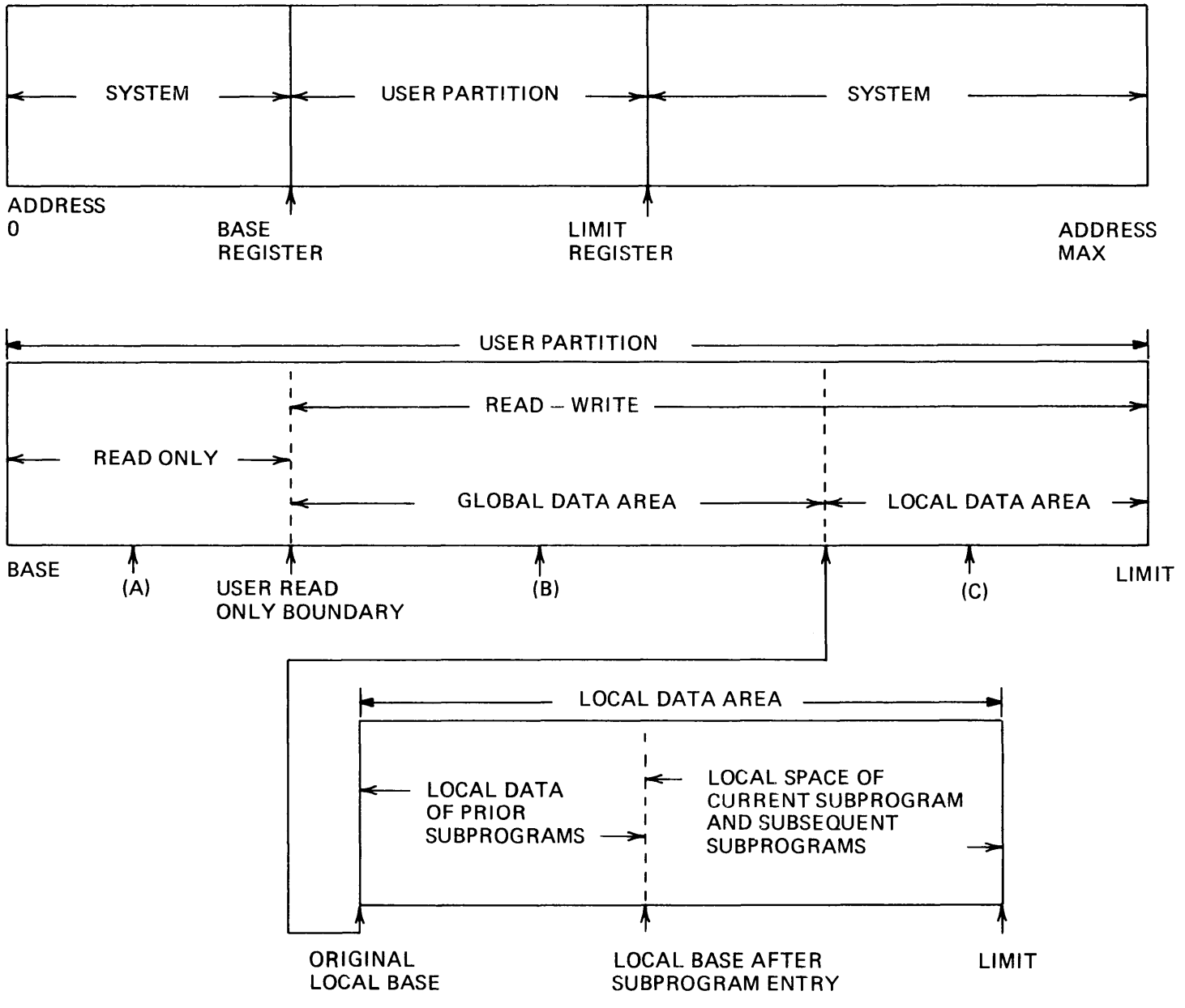

Figure 7. Typical Memory Segmentation, Control Memory

memory protection, whereby base and limit registers are provided to prevent references outside the the problem program's space, or to prevent modification of read-only information such as executable code and constants. This is accomplished by subdividing each overlay into segments according to class of memory required. Figure 7 illustrates a segmentation pattern for control memory; the corresponding pattern for parallel memory is similar to that of the read-write user partition.

The overlay loader is invoked by the BSP presence semaphore mechanism. Each phase is assigned to a unique hardware presence semaphore, which can be efficiently tested by special instructions. The linker inserts such a test instruction preceding each call or I/O reference to a subroutine or I/O unit in another possibly nonresident phase. If the phase is not present, the test will cause an interrupt invoking the BSP overlay loader software, which will load the phase (possibly first writing out SAVE data to be overlayed), set the presence semaphore and then allow the program to resume. Presence tests for a present phase will allow the program to continue without an interrupt.

The BSP overlay mechanism thus eliminates much of the overhead associated with conventional demand-paged systems by pregrouping related routines (working sets), thus minimizing the number of presence faults and the access time to locate individual pages. Finally, it should be noted that the superior access times of the CCD file memory further minimize the delay for overlay loading.

The BSP loader also assists in reducing delays of interprogram transition. If the running program does not require all of processor memory, a subsequent program will be preloaded into the available space.

9. SUMMARY

The Master Control Program for the BSP provides a fully integrated and unified user interface to a functionally distributed system. In so doing, it provides a new level of convenience for the supercomputer installation, which heretofore was often forced to contend with ad hoc interfaces or commit extensively to in-installation-developed operating system software.

The BSP MCP also reaffirms a commitment to efficiency and system balance. By carefully distributing functions, the MCP is able to provide user convenience where it is most needed, as in program development, while retaining the efficiency of low software overhead in the time-critical domain of scientific program execution.

Finally, the BSP MCP heralds a new trend in hardware-software tradeoff, in that a significant "operating system" function, so-called physical I/O, has been completely off-loaded into the input-output controller.

Together, these developments demonstrate the synergistic effect of a combined hardware-software-systems approach to a particular problem, such as in this instance, the problem of ultra high-speed numerical computation.

Burroughs Corporation

61391 2/78 Printed in U. S. A.