The HOCA Operating System Specifications*

Özalp Babaoğlu
Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

August 26, 1996

* Partial support for this work was provided by the National Science Foundation under Grants No. MCS82-10356 and MCS81-03605.

Acknowledgments

The HOCA operating system implementation project has evolved over the years at Cornell through contributions of Gregory Andrews, Thomas London, Charles Moore, Alan Shaw and David Wright. We are grateful to them. Earlier versions of the project were designed to be written in XPL and run on the CS983 computer simulator. The current version of HOCA is designed to be implemented in C and executes on the CHIP host computer that is simulated on a UNIX† system.

Rogerio Drummond and Mimi Bussan implemented the CHIP simulator and console interface. Gilbert Neiger was responsible for the first realization of HOCA for the new machine in C. Gilbert Neiger and Rogerio Drummond were also instrumental in revising this specification document to reflect the new host and programming environment.

Finally, we would like to thank the many students in our courses who have helped debug the project design specifications and the simulator environment.

---

† UNIX is a Trademark of AT&T Bell Laboratories.

Chapter One
Introduction

This document describes the specification and some hints for design and construction of HOCA†, a simple operating system that has been designed as an educational tool to supplement formal operating system ideas through implementation. The system is level structured and incorporates most of the concepts found in modern operating systems. At Cornell, the HOCA implementation project accounts for the entire workload in a junior-level, two credit course called *Practicum in Operating Systems* which is a companion to the operating systems theory course.

The host machine for the project is a PDP-11 look-alike computer called CHIP (Cornell Hypothetical Instructional Processor). A description of the CHIP architecture and programming environment can be found in *Documentation for the CHIP Computer System*, Ö. Babaoğlu *et al.*, Cornell University, Department of Computer Science Tech. Report No. 83-584.

HOCA is designed to be implemented in three phases. The first phase is intended to familiarize students with the C programming language, UNIX and the CHIP simulator. It involves implementing two modules that are later used in the nucleus to implement the process queue and the semaphore abstractions. The next phase is to build the nucleus—the routines that implement the notion of asynchronous sequential processes, a pseudo-clock and the synchronization primitives. The third phase involves implementing another software level (called the ‘‘support level’’) that creates an environment with virtual memory, input/output devices and virtual synchronization operations that is suitable for the execution of user programs.

† *Hoca* (pronounced *hod-ja*), is a Turkish word meaning ‘‘schoolmaster’’ or ‘‘giver of good advice.’’

Chapter Two
Implementing Modules in C

HOCA is implemented in the C programming language. Unfortunately, C has no support for *modules*. However, some of the important aspects of modules can be approximated using the separate compilation mechanisms of C. In this chapter, we briefly describe how to achieve this. Some other conventions in organizing your files will also be introduced.

Among the set of files that make up a C program we distinguish three different varieties and give them distinct extension names. Files having the extension ‘‘.c’’ (for *C code*) contain the *definitions* for a given module. These include constants, macros, types, variables and functions. Of these definitions, those that may be accessed by other modules are *declared* in a ‘‘.e’’ (for *export*) file. Constants, macros and types that are used by a number of modules and don’t belong to any module in particular are collected in ‘‘.h’’ (for *header*) files.

A definitions file should have its contents in the following order:

1. Inclusion of header files.
2. Inclusion of declarations from other modules that are being *imported*.
3. Definitions of macros, types and variables to be exported.
4. Definitions of macros, types and variables strictly local to the module.
5. Definitions of functions to be exported.
6. Definitions of functions strictly local to the module.

Variables and functions that are strictly local to the module should be defined with the **static** attribute so that they will not be visible from other modules (files). The only exception to this rule is the function *main()* which should not be exported but also should not be defined using **static** since it is the entry point to the program. In our examples, we will use the word ‘‘HIDDEN’’ rather than **static** to better capture our intended use of the mechanism.

You will use the **#include** mechanism of the C preprocessor to include the header files and the declarations of other modules (‘‘.e’’ files) that are being imported by the current module.

An export file corresponding to a definitions file should have the following structure:

1. Constants, macros and types from item 3 of above.
2. Declarations of the variables from item 3 of above.
3. Declarations of the functions from item 5 of above.

Finally, a header file contains:

1. Literals introduced for program readability. For example,

```
#define TRUE        1
#define FALSE       0
#define PAGESIZE  512
#define HIDDEN      static
```

2. Utility macro definitions. For example,

```
#define ODD(x)      (x & 01)
```

3. Parameters of the program or set of modules. For example,

```
#define MEMSIZE   (32 * PAGESIZE)
#define MAXPROC   20
```

Note the use of capital letters for names of constants and macros in order to distinguish them from variables and functions.

You are expected to adhere strictly to these conventions in the code you produce. A large percentage of your style grade will be based on the structuring of your files according to these conventions.

An example of a FIFO queue module illustrating some of these conventions can be found in Appendix 1.

Chapter Three
Program Grading Guide

1. Policies

The assignments will be graded according to a 0-100 point scale. The correctness of your solution and the style/clarity of programming will weigh equally towards your score. Late assignments will have 5 points deducted for each day that they are late (including weekends). Consequently, you are encouraged to turn in a nearly correct program on time rather than spend a few additional days perfecting it.

The following principles will be used in grading programs. The first two items have to do with style, and the final item deals with correctness. It goes without saying that a program that does absolutely nothing correctly but has good style is not worth much.

2. Program Structure

(a) Global structure: The file structure should conform to the description given in Chapter 2. Namely, each ‘‘.c’’ file that implements a module should have an associated ‘‘.e’’ file if there are any declarations visible to other modules. Definitions common to a set of modules (and only these) should be collected in ‘‘.h’’ files.

(b) Structure within files: The order of includes, declarations and definitions should follow the one given in Chapter 2. Your programs should be properly indented. Use *The C Programming Language* textbook as a guide for this. Use one ‘TAB’ character for each level of indenting. Do not indent using blanks. Use white space (blank lines) to separate groups of related (commented) statements. The various sections of a file (items 1 through 6 for ‘‘.c’’ files as discussed in Chapter 2) should also be separated by blank lines.

(c) Constants: There should be no constants embedded in your code. All constants should be given mnemonic names using the **#define** mechanism and references to them should be through these names. Here, we do not mean

#define FIVETWELVE 512

but rather

#define PAGESIZE 512

The only allowable exception to this rule is the use of the constants 0 and 1.

3. Layout of Comments

Every module should have a header comment as follows:

(a) Description of the data structure (or abstraction) implemented and maintained by the module.

(b) The module invariant. This states (in terms of the module variables) what remains true throughout the execution of the module by calls to its externally visible functions. Here, you should be as formal as you can (using mathematical notation) without being cumbersome. An English description of the invariant is acceptable if it would require clumsy notation to express formally.

(c) A brief description of how the module implements the stated abstraction.

Every function (externally visible or local) of the module should have a header comment that describes what the function does.

In commenting the code, use a comment for a block of statements. Here, we do not mean C blocks ({...}), but groups of related statements. In fact, a single C block may contain several commented sub-blocks and/or be part of a larger commented super-block. Comments should be aligned with the indentation level of the statements to which they apply.

Use a comment as the first line of the two clauses of an **if** statement to clarify the condition under which the clauses are executed. For example,

```
if (semd_h == NULL || semd_h—>s_next == semd_h) {
      /* list has one or zero elements */
            ...
} else {
      /* list has more than one element */
            ...
}
```

Many variables of a program or function have meanings that remain unchanged (invariant) with execution. The declaration of these variables should always be commented by including a brief statement of their meaning on the right side of the same line.

Declaration of variables used in different contexts or used only to hold temporary values need not have a comment. Use of names such as ‘‘temp,’’ ‘‘i,’’ ‘‘k,’’ etc. for them will relay this fact.

Try to strike a balance between undercommenting and overcommenting. A well-commented program is not one that has a comment for each line of code. Do not construct comments that simply restate what is obvious from the code. For example, the comment ‘‘assign zero to p’’ is useless when referring to the statement ‘‘p = 0.’’ Your comments should be a high level description of what is going on.

4. Correctness

The following aspects of correctness will be considered in computing project grades:

(a) Incorrect initialization.

(b) Program or function does not follow the required specifications.

(c) Program of function does not handle special cases. You should make no assumptions about the valid range of input data to a function unless it is stated in the specification of the problem. The entire possible input data range should be handled in some reasonable manner.

(d) Inefficient implementation.

Chapter Four

Phase One: Process and Semaphore Queue Modules

1. Introduction

In this phase you will write two simple modules to create and manipulate queues of processes and semaphores. The next phase (nucleus) will use these modules to carry out some of its tasks. The code you produce will be ‘‘handed in’’ by running the **submit** program that will make the named files visible to us for grading.

You are strongly encouraged to create four subdirectories in your home directory. Three of them will contain the code for each of the three phases, and the fourth, called *h*, will contain the included (i.e., ‘‘.e’’ and ‘‘.h’’) files. Appendix 2 contains the listing of two files defining certain hardware-related constants and types for the CHIP computer. These will be very useful for you. The two files are available in the directory /stage/projects4/CHIP/hoca/h under the names *types.h* and *const.h*. You should copy them into the *h* subdirectory of your account and make additions (deletions) as you need them. For this assignment, you will also need the following type definitions.

```
/* process table entry type */
typedef struct proc_t {
        struct proc_t   *p_next;        /* pointer to next entry */
        state_t         p_s;            /* processor state of the process */

        /* plus other entries to be added by you later */

} proc_t;



/* semaphore descriptor type */
typedef struct semd_t {
        struct semd_t   *s_next,        /* next element on the queue */
                        *s_prev;        /* previous element on the queue */
        int             *s_semAdd;      /* pointer to the semaphore proper */
        proc_t          *s_procq;       /* pointer to the tail of the queue of */
                                        /* processes blocked on this semaphore */
} semd _t;
```

In coding, be sure to follow the conventions described in Chapters 2 and 3. In particular, remember that each module must have a header describing the module, its invariant and the data structures used. Each function definition should be preceded by a precise description of *what* it does and not *how* it is implemented.

You are strongly encouraged to use the **make** program to maintain your code. The manual page for it is on the system (type **man make**). Use the file *makefile* in the directory /stage/projects4/CHIP/hoca/phase1 as a template in constructing yours. Note that the command lines associated with a made product have to begin with a ‘TAB’ character (*not* blanks).

2. Process Queue Module

This module creates the abstraction of queues of processes. We suggest the following implementation. The elements in each of the process queues come from the array *procTable[MAXPROC]* of type *proc_t* (shown above). The unused elements of this array (i.e., the elements currently not in any process queue) are kept on a NULL-terminated simple linked list headed by the variable *procFree_h*. We will refer to this list as the procFree list. The variables *procTable* and *procFree_h* are local to the process queue module (i.e., declared as HIDDEN).

The module should have the following externally visible functions:

insertProc(tp,p)
    proc_t **tp, *p;

> Insert the element pointed to by *p* into the process queue where *tp* is a pointer to the pointer to the tail (last element). Note the double indirection through *tp*. Update the tail pointer (**tp*) accordingly.

proc_t *
removeProc(tp)
    proc_t **tp;

> Remove the first element from the process queue whose tail-pointer (*not* tail) is pointed to by *tp*. Return NULL if the queue was initially empty, otherwise return the pointer to the removed element. Update the pointer to the tail of the queue if necessary.

proc_t *
outProc(tp,p)
    proc_t **tp, *p;

> Remove the process table entry pointed to by *p* from the queue whose tail-pointer is pointed to by *tp*. Update the pointer to the tail of the queue if necessary. If the desired entry is not in the defined queue (an error condition), return NULL. Otherwise, return *p*. Note that *p* can point to any element of the queue (not necessarily the head element).

proc_t *
allocProc()

> Return NULL if the procFree list is empty. Otherwise, remove an element from the procFree list and return a pointer to it.

freeProc(p)
    proc_t *p;

Return the element pointed to by *p* into the procFree list.

initProc()

Initialize the procFree list to contain all the elements of the array *procTable*. Will be called only once during data structure initialization.

3. Active Semaphore List Module

This module creates the active semaphore list (ASL) abstraction. A semaphore is *active* if there is at least one process blocked on it. We suggest the following implementation for this module. The head of the active semaphore list is pointed to by the variable *semd_h*. The elements in this list come form the array *semdTable[MAXPROC]* of type *semd_t* (defined in the introduction). The unused elements of the array *semdTable* are kept on a NULL-terminated free list headed by the variable *semdFree_h*. The variables *semdTable*, *semd_h* and *semdFree_h* are all local to the module. The ASL is a doubly-linked queue of semaphore descriptors sorted by semaphore address.

This module should have the following externally visible functions:

insertBlocked(semAdd,p)
int *semAdd;
proc_t *p;

Insert the process table entry pointed to by *p* at the tail of the process queue associated with the semaphore whose address is *semAdd*. If the semaphore is currently not active (there is no descriptor for it in the ASL), allocate a new descriptor from the free list, insert it in the ASL (at the appropriate position), and initialize all of the fields. If a new semaphore descriptor needs to be allocated and the free list is empty, return TRUE. In all other cases return FALSE.

proc_t *
removeBlocked(semAdd)
int *semAdd;

Search the ASL for a descriptor of this semaphore. If none is found, return NULL. Otherwise, remove the first process table entry from the process queue of the appropriate semaphore descriptor and return a pointer to it. If the process queue for this semaphore becomes empty, remove the descriptor from the ASL and insert it in the free list of semaphore descriptors.

proc_t *
outBlocked(p)
    proc_t *p;

> Remove the process table entry pointed to by *p* from the queue associated with the appropriate semaphore on the ASL. If the desired entry does not appear in any of the process queues (an error condition), return NULL. Otherwise, return *p*.

proc_t *
headBlocked(semAdd)
    int *semAdd;

> Return a pointer to the process table entry that is at the head of the process queue associated with semaphore *semAdd*. If the list is empty, return NULL.

initSemd()

> Initialize the semaphore descriptor free list.

Note that the ASL module will make calls to the process queue module to manipulate the blocked process queues associated with semaphores. You should add to the process table entry and semaphore descriptor types whatever fields you feel are necessary to render the implementation of these functions efficient. Also, feel free to define and implement functions in addition to those above that you need.

4. Testing

You should compile the two modules you write separately using the **-c** option to **pcc** as follows:

> **pcc -c** asl.c
> **pcc -c** procq.c

This will generate the two files *asl.o* and *procq.o*. The test program to exercise your two modules is /stage/projects4/CHIP/hoca/phase1/p1test.c. You should *link* to the compiled version of this program (*p1test.o* in the same directory) by saying

> **ln** /stage/projects4/CHIP/hoca/phase1/p1test.o p1test.o

in your phase1 directory. You can load the test program along with your modules through

> **pcc** asl.o procq.o p1test.o **-o** p1test

To run the generated program *p1test* on CHIP, type

> **chip** p1test

The test program reports on its progress by adding messages to the buffers *okbuf* and *errbuf*. The second buffer is written only if an error is detected. You can display the contents of these buffers by inserting their names on the trace list of the CHIP console

> **ti** okbuf:+514/a errbuf:+100/a

As the program executes (when you type **run**), the progress messages will appear on the screen in ASCII format for you to observe. Use this assignment to familiarize yourself with the various features of the CHIP simulator. After debugging your modules, run the test program to completion and then write the contents of the dynamic screen (containing the entire sequence of progress messages in the two buffers indicated above) to a file by using the **fout** command of the simulator. Submit this file along with your code files for grading.

Chapter Five
Phase Two: The Nucleus

1. The Nucleus

The nucleus described below is similar to the T.H.E. system constructed by Dijkstra, and to the one discussed in lecture. The purpose of the nucleus is to provide an environment in which asynchronous sequential *processes* exist, as well as to provide synchronization primitives for these processes. This implies that the nucleus must do low-level CPU scheduling. In addition, the nucleus provides facilities for ‘‘passing up’’ conditions such as program traps, memory management traps and certain system call traps to the next level in the level structure.

The abstract machine implemented by the nucleus is described in the following sections. The instructions implemented by the nucleus are invoked when a process executes the corresponding SYS instruction. In addition, various interrupts and traps cause the nucleus to take actions. In particular, the nucleus transforms clock and I/O interrupts into V operations on semaphores; SYS, program, and memory management traps are changed into procedure calls.

The nucleus has frequent occasion to deal with regions of memory containing a 24-byte copy of the processor state, i.e., the general purpose registers, SP, PC, PS1, PS2, the STA and the STL registers. This will be referred to as a *processor state*.

2. Assumptions

In writing your nucleus, you may assume that no more than 20 (this constant should be equivalent to the mnemonic ‘‘MAXPROC’’) processes will exist at any time. All addresses the nucleus deals with may be assumed to be physical (as opposed to virtual). The nucleus should preserve the state of a process; if a process has memory mapping enabled when it is interrupted or executes a nucleus-implemented (SYS) instruction, then it should still be in that state when it continues executing. Your nucleus should guarantee finite progress. Consequently, every ready process will have an opportunity to execute. You are encouraged to implement CPU multiplexing using a simple round-robin scheduler with time slice value not greater than 5 milliseconds.

3. Handling Interrupts and Traps

After the nucleus has initialized itself when the machine is first turned on, the only mechanism for entering it is through an interrupt or trap. As long as there are processes to run, the machine is executing instructions on their behalf and temporarily enters the nucleus long enough to handle interrupts and traps when they occur.

I/O interrupts should function as V operations on the semaphore associated with the appropriate device. Furthermore, the the contents of the device’s Status register and, if appropriate, Length register, should be saved; this is the I/O operation’s *completion status*. It is only necessary to save the most recent completion status for each device. If a process is waiting for I/O for the interrupting device (see SYS8 below), the completion status is passed directly to that process and need not be saved.

The Interval Timer of the CHIP will be used both for CPU scheduling (enforcing the time slice) and implementing the *pseudo-clock*. This is nothing more than a special semaphore that is V'ed by the nucleus every 100 milliseconds. This mechanism will be used later to implement a general delay facility for processes.

Traps generated by certain SYS instructions cause the nucleus to perform some service on behalf of the process executing the instruction. The specification of these services is detailed in the next section. SYS traps other than the ones listed below, as well as all program traps and memory management traps may be ''passed up'' by the nucleus to the process responsible for them. From the point of view of a process, this mechanism simulates a software interrupt by saving the old processor state in memory and loading a new state from a different place in memory. These two memory addresses (one for the old state and another for the new state) together constitute the process's *trap state vector*. This is initialized by a process when it executes a SYS5 instruction. The level above the nucleus will provide the handlers for these types of traps.

4. New Instructions

The following instructions (SYS1 - SYS8) are handled directly by the nucleus. They are executable only by processes running in kernel state. The nucleus should confirm this and in case of execution by a user state process, the request should be treated as a privileged instruction program trap without executing the requested operation. That is, the appropriate SYS instruction will not be executed and the process's *program trap state vector* will be used as outlined below.

Create_Process (SYS1)

When executed, this instruction causes a new process, said to be a *progeny* of the first, to be created. R4 will contain the address of a processor state area at the time this instruction is executed. This processor state should be used as the initial state for the newly created process. The process executing the SYS1 instruction continues to exist and to execute. If the new process cannot be created due to lack of resources (for example no more entries in the process table), an error code of −1 is returned in R2. Otherwise, R2 contains zero upon return.

Terminate_Process (SYS2)

This instruction causes the executing process to cease to exist. In addition, any progeny of this process are terminated (and, by implication, the progeny of the progeny, etc.). Execution of this instruction does not complete until *all* progeny are terminated.

Verhogen (V) (SYS3)

When this instruction is executed, it is interpreted by the nucleus as a V operation on the semaphore whose address is in R4 at the time the instruction is invoked. Semaphores are assumed to be full word integers.

Passeren (P) (SYS4)

When this instruction is executed, it is interpreted by the nucleus as a P operation on the semaphore whose address is in R4 at the time the instruction is invoked.

Specify_Trap_State_Vector (SYS5)

When this instruction is executed, it supplies three pieces of information to the nucleus:

- The type of trap for which a *trap state vector* is being established. This information will be placed in R2 at the time of the call, using the following encoding:

  0 - program trap
  1 - memory management trap
  2 - SYS trap

- The area into which the processor state (the old state) is to be stored when a trap occurs while running this process. The address of this area will be in R3.
- The processor state area that is to be taken as the new processor state if a trap occurs while running this process. The address of this area will be in R4.

The nucleus, on execution of this instruction, should save the contents of R3 and R4 (in the process table entry) to pass up the appropriate trap when (and if) it occurs while running this process. When that happens, the nucleus stores the processor state at the time of the interrupt in the area pointed to by the address in R3, and sets the new processor state from the area pointed to by the address given in R4.

Each process may execute a SYS5 instruction *at most* once for each of the three trap types. An attempt to execute a SYS5 instruction more than once per trap type by any process should be construed as an error and treated like a SYS2.

If a trap occurs while running a process which has not executed a SYS5 instruction for that trap type, then the nucleus should treat the trap like a SYS2 (Terminate_Process).

Note that only instructions SYS1 to SYS8 are treated as *instructions* by the nucleus. The other SYS instructions (SYS9 to SYS255) are passed up as SYS traps.

Get_CPU_Time (SYS6)

When this instruction is executed, it causes the CPU time (in microseconds) used by the process executing the instruction to be placed in R2 and R3. The high order word is placed in R2 and the low order in R3. This means that the nucleus must record (in the process table entry) the amount of processor time used by each process.

Wait_for_Clock (SYS7)

This instruction performs a P operation on the pseudo-clock semaphore. This semaphore is V'ed every 100 milliseconds by the nucleus.

Wait_for_IO_Device (SYS8)

This instruction performs a P operation on the semaphore that the nucleus maintains for the I/O device whose number is in R4. All possible devices for the CHIP are assigned unique numbers in the range 0-14 as shown in the *const.h* file of Appendix 2. The appropriate device semaphore is V'ed every time an interrupt is generated by the I/O device.

Once the process resumes after the occurrence of the anticipated interrupt for printer and terminal devices, R2 should contain the contents of the Length register for the appropriate device and, for all devices, R3 should contain the contents of the device's Status register. These two registers constitute the I/O operation completion status (see above), and may have already been stored by nucleus. This will occur in cases where an I/O interrupt occurs before the corresponding SYS8 instruction.

5. Initial Startup

It is *your* responsibility to test your nucleus. We will supply a C function called *test()* (source is in /stage/projects4/CHIP/hoca/nuctest/test.c for you to look at and the object code in /stage/projects4/CHIP/hoca/nuctest/test.o) that exercises some of the primitives to help us evaluate your nucleus. You should link (DO NOT COPY) to the file /stage/projects4/CHIP/hoca/nuctest/test.o in your nucleus directory and load it along with the rest of your nucleus. You can do this by compiling each file of your nucleus with the **-c** flag to **pcc**. This will produce a ‘‘.o’’ file corresponding to the ‘‘.c’’ file. To load all these files along with the test program, you should say

pcc *your nucleus .o files* test.o -o nucleus

Then you can run the file *nucleus* on CHIP. Remember to declare the test function as ‘‘external’’ in your program by saying

extern int test();

before you refer to it. You can initialize a variable to contain the starting address of a function by simply naming the function in the right hand side of an assignment statement. For example, to initialize a processor state area so that the function *test()* is invoked when it is loaded as the new state, you can say

```
state_t new;
      ...
new.s_pc = (int)test;
```

where *s_pc* is the program counter field of the processor state structure.

When your nucleus comes alive, it should figure out how much physical memory there is on the machine (this amount will always be a multiple of the page size). This can be done by examining the STL register, which contains an address one full page above the bottom of memory. (SP will not contain an address just below the bottom of memory because a function (*main()*) has already been called.) The *new* areas for the traps and interrupts should be initialized so that the nucleus stack starts to grow from the very bottom of physical memory. We suggest that the nucleus be allocated one page of stack space, but that this size be easily modifiable if needed. After it has completed initializing the necessary data structures, the nucleus should create a single process running the code of the function *test()* in a state identical to the nucleus except with all interrupts unmasked. The SP of the new state for this process should be immediately above the nucleus stack. The size of the test process stack should also be one page. This test process will exercise your nucleus and generate some output on the device printer0 (mapped to a file called ‘‘printer0’’ in the current directory) to indicate its progress. You will have to submit this file along with the source of the nucleus for us to evaluate your nucleus.

6. Accessing Machine Registers Through C

Note that the ‘‘arguments’’ to the various nucleus-implemented instructions are passed through general purpose registers (in particular, R2, R3 and R4). Consequently, to invoke these nucleus services a process must be able to initialize the registers with the actual values of the arguments before issuing the appropriate SYS instruction. This can be accomplished easily through C by declaring the following local register variables in the first line of a function where the nucleus calls are to be made:

```
test()
{
        register int r4, r3, r2;
        ...
        r4 = devno;
        DO_WAIT_IO();           /* defined to be SYS8() */
        if (r3 == NORMAL) {
                num = r2;
                ...
        } else {
                ...
```

By convention, the C-compiler allocates the first three local register variables to the machine registers R4, R3 and R2, in that order. Therefore, in the code fragment above, we have assigned the variable names r2, r3 and r4 to the machine registers R2, R3 and R4, respectively. Any manipulation of these variables is equivalent to the manipulation of the corresponding register. It is important that this declaration be the first local register declaration and the order of variable names be as given.

When the nucleus finally assumes control after a SYS instruction, the machine registers contain values that were initialized from the SYS trap new area. The register values prior to the SYS trap are stored in the old area associated with this trap. Therefore, the code within the nucleus implementing a certain service can obtain the values of the arguments being passed to it by simply looking at the locations corresponding to the appropriate registers in the SYS trap old area.

7. Termination

The nucleus should detect certain very simple deadlock states. If all of the existing processes are blocked due to P operations on semaphores not associated with I/O devices or the pseudo-clock, the system should shut down with an appropriate message. The nucleus should also shut down (normally) when the last process in the system terminates.

Chapter Six
Phase Three: The Support Level

1. Introduction

For the next level of your operating system, you are to design and implement support for virtual memory and other facilities that can be used by a simple timesharing system. The timesharing system will support up to five terminals. Associated with each terminal is a single process, called a T-process, that will execute programs submitted through that terminal. In addition, several system processes will be required in order to handle paging, the pseudo-clock and disk I/O.

We want a way for each T-process to request system services without changing the nucleus or compromising system security. This suggests implementing new SYS operations. These are outlined in the next section.

However, several problems need to be addressed when providing this support:

(a) I/O — a process that can issue I/O operations could specify any memory location for data transfer and can thus overwrite any location it wishes.
Solution: make sure the page frame containing the device registers is not accessible to the T-processes.

(b) Nucleus implemented SYS's — by specifying random locations as, say, semaphores, a process could alter any location it liked.
Solution: run the T-processes in user mode, since they are then prohibited from issuing nucleus implemented SYS's. (This is why the nucleus specification contained this restriction.)

(c) Arbitrary memory accesses using stores, moves, etc.
Solution: use memory mapping hardware to give each process a private address space disjoint from both the support level and other T-processes. (Actually, the specifications do provide for a segment to be shared by all T-processes, and a malicious process could interfere with others that were trying to, say, synchronize. However, this is not a security hole, merely a nuisance.)

Each T-process will run in user state, with interrupts unmasked, and will have an address space consisting of two segments: segments 2 and 3. (Recall that the CHIP memory management supports four segments numbered 0, 1, 2, and 3.) Segment 3 is shared by all of the T-processes, segment 2 is unique for each. Thus T-processes communicate and synchronize by using segment 3.

T-processes cannot be assumed to be trustworthy. Thus, it will be prudent for you to construct this level so that it protects itself from T-processes. For one thing, this will make it easier to debug; T-processes will be stopped before they completely destroy the integrity of the system.

The system processes of this level will run with memory mapping on, in kernel mode and with interrupts unmasked. The code and data for these processes will reside in segments 0 and 1 together with the nucleus. These processes will always be resident in memory; the pages of segments 0 and 1 will not be paged out. A missing segment or missing page memory management trap for an address in segments 0 or 1 while a system process is running will denote an error condition.

Since this is a level-structured system, it should not be necessary for you to make any changes to your nucleus in order to complete this new level. In fact, this level should be able to run on top of any correct nucleus. A nucleus that we believe to be correct will be available for your use, should you decide not to use the one you built during Phase 2. The source files for our nucleus are in the directories /stage/projects4/CHIP/hoca/nucleus and /stage/projects4/CHIP/hoca/h. You can look at these files using the **seenuc** command (this is just like **more**). The object file for the nucleus that you can link with your support level is the *nucleus.o* file in the same directory. The details of how to load the nucleus and support level are covered in the next chapter.

Recall that your nucleus started running a process named *test* after it had completed initialization. Thus, you should name the code that is to receive control to initialize this level *test*.

2. New Instructions

The following ‘‘instructions’’ are supported by this level. T-processes request these operations by executing the appropriate SYS instructions. Recall that executing a SYS causes control to be transferred to the nucleus SYS handler. But, if the appropriate SYS5 instruction has been executed for the process that caused the SYS trap, then a higher level trap handler can receive control. Thus, among the first things a T-process should do after it has been created are three SYS5 instructions (one for each type of trap it can generate). As the T-processes themselves are not trustworthy, when the support level is creating a T-process, it should start a process that does the appropriate SYS5's and some other initialization (more on this in the next chapter) and then does a LDST to transfer control to the ‘‘real’’ T-process.

The pseudo-instructions supported by this level are:

Read_from_Terminal (SYS9):

> Requests that the invoking T-process be suspended until a line of input has been read from the associated terminal. The data read should be placed in the virtual memory of the T-process executing the SYS9, at the (virtual) address given in R3 at the time of the call. The count of the number of characters actually read from the terminal should be available in R2 when the T-process is continued. An attempt to read when there is no more data available should return the negative of the ‘‘End of Input’’ status code in R2. If the operation ends with a status other than ‘‘Successful Completion’’ or ‘‘End of Input’’ (as described above), the negative of the Status register value should be returned in R2. Any negative numbers returned in R2 are ‘‘error flags.’’

Write_to_Terminal (SYS10):

> Requests that the T-process be suspended until a line (string of characters) of output has been written on the terminal associated with the process. The virtual address of

the first character of the line to be written will be in R3 at the time of the call. The count of the number of characters to be written will be in R4. The count of the number of characters actually written should be placed in R2 upon completion of the SYS10. As in SYS9, a non-successful completion status will cause an error flag to be returned instead of the character count.

V_Virtual_Semaphore (SYS11):

Performs a V operation on a semaphore whose virtual address is in R4 at the time of the call.

P_Virtual_Semaphore (SYS12):

Performs a P operation on a semaphore whose virtual address is in R4 at the time of the call. You may *not* assume that the only virtual addresses passed in R4 for SYS11 and SYS12 will refer to segment 3.

Delay (SYS13):

On entry, R4 contains the number of seconds (wall clock time) for which the invoker is to be delayed. The caller is to be delayed at least this long, and not substantially longer. Since the nucleus controls low-level scheduling decisions, all that you can ensure is that the invoker is not dispatchable until the time has elapsed, but becomes dispatchable shortly thereafter.

Disk_Put (SYS14) and **Disk_Get** (SYS15):

These two instructions are optional and will be worth extra credit.
These instructions are provided to perform *synchronous* I/O on the DISK0 device. Some form of seek reduction algorithm should be implemented in the disk driver process in order to facilitate high disk utilization. On entry, R3 should contain a virtual address from (to) which the disk transfer is to be made. R4 should contain the disk track number and R2 the sector within the track to (from) which the data transfer will be made. The corresponding write (read) operation of the 512-byte data is initiated, and only after it has completed is the T-process resumed. The number of bytes actually read or written replaces the contents of R2. As for the SYS9 and SYS10 instructions, the negative of the Status register is returned in R2 instead of the byte count if the operation does not complete successfully. Note that the disk controller of the CHIP assumes that the memory buffer involved in the I/O is a *physical* address.

Get_Time_of_Day (SYS16):

Returns the value of the time-of-day clock in R2 (high word) and R3 (low word).

Terminate (SYS17):

> Terminates the T-process. When all T-processes have terminated, your operating system should shut down. Thus, somehow the ‘‘system’’ processes created in the support level must be terminated after all five T-processes have terminated. Since there should then be no dispatchable or blocked processes, the nucleus will halt.

3. Implementing the Virtual Memory

Since all T-processes share segment 3, all references by T-processes to addresses in segment 3 should reference the same data. References to addresses in segment 2 should reference different data (pages) for each T-process. The important implication of the shared segment 3 is that it doesn’t belong to any process. It also means that several page faults may occur for a single page simultaneously, since several processes can reference the same shared page.

The DRUM0 device will be used for paging. Pages of segment 2 of each T-process and the pages of the common segment 3 will be associated with addresses on the DRUM0 device where they will reside while the T-process exists. The page replacement algorithm you implement can be any reasonable one of your choosing.

4. Handling Traps in T-Processes

The support level should take some sane action if a T-process causes a trap that it was not expecting. A non-exhaustive list of such traps includes:

- Memory management traps for pages other than those in segments 2 or 3.
- Program traps.
- Attempts to execute undefined SYS’s.
- Passing unreasonable parameters to SYS instructions implemented in this level.

The response to an unreasonable action on the part of a T-process should probably be to terminate it — treat the action as equivalent to a SYS17. Note that T-processes should be run in user state with memory mapping on, and therefore should not be able to address directly either the nucleus or the support level address space. Therefore, they cannot subvert the functions implemented by the nucleus or this level by directly modifying instructions or data structures associated with them. Nor can they execute privileged instructions such as LDST, HALT, etc. Also note that the handlers for the T-processes and the handlers for the system processes might be different due to the distinct nature of these processes. More on this in the next chapter.

5. Initialization

A process and virtual memory should be created for each of the terminals. Each T-process you create should start executing at address 0100000 (segment 2, page 0, byte 0) in its virtual memory. The first thirteen words of segment 2 should have been initialized with the contents of the following C array:

```
int bootcode[ ] = {
      0012703,
      0100032,
      0000406,
      0060203,
      0022702,
      0000200,
      0003402,
      0112723,
      0000012,
      0104411,
      0005702,
      0002367,
      0000410
};
```

This array declaration is available as the file /stage/projects4/CHIP/hoca/boot/bootcode.c for you to copy. These octal numbers happen to correspond to the object code of the following ‘‘bootstrapping loader’’ written in assembly language (this is presented here only for your information):

```
            mov     $_out,r3
            jbr     L20001
L20003:     add     r2,r3
            cmp     $200,r2
            jle     L20001
            movb    $12,(r3)+
L20001:     sys     9.
            tst     r2
            jge     L20003
            jbr     _exec          / skip past the 8-word header
            .globl  _out
_out:
            .=.+16.                / space for the header
_exec:
```

It causes a program to be read in from the terminal associated with the current T-process, and then executed.

6. Testing the Support Level

There are five test programs for testing the support level, including one to test disk I/O for those who choose to implement it. These programs are available in the /stage/projects4/CHIP/hoca/suptest directory under the names *termin0*, ... *termin4*; You should link to these files in your phase3 directory. Don’t copy them because we might find the need to change these files. The C code for the test programs are also available in the same directory under the names *t0.c*, ... , *t4.c* for you to look at. The five programs are:

1) termin0 — contains general tests for various SYS's, paging.
2) termin1 — tests disk I/O
3) termin2 — tests the swapper and paging
4) termin3 & termin4 — must run together, test virtual semaphores.

You can write your own support level test programs before attempting to run ours. The C program you write to test the support level will have to be compiled using the **maketest** program rather than **pcc** (so that the code will be relocated to live in segment 2). The **maketest** program takes a sequence of C files as argument and produces an assembler listing and an object file for each. You will have to rename the object file produced to be one of *termin0 ... termin4* before running the simulator.

Chapter Seven
Notes on the Support Level

1. Compiling and Loading

The files of the nucleus and the support level should be compiled separately using the **-c** option and then loaded as follows:

pcc *nucleus .o files* /stage/projects4/CHIP/hoca/lib/crt1.o *support level .o files* **-o** hoca

This will produce an executable file called *hoca* with the code and data loaded as shown in Fig. 1. You must include the /stage/projects4/CHIP/hoca/lib/crt1.o file in the position shown above (do not copy or link to this file).

The C-compiler generates three areas as a result of a program compilation:

(1) The *text* area. This is where the instructions of the program are put in.

(2) The *data* area. This is where the program variables with an *initialized* value are placed.

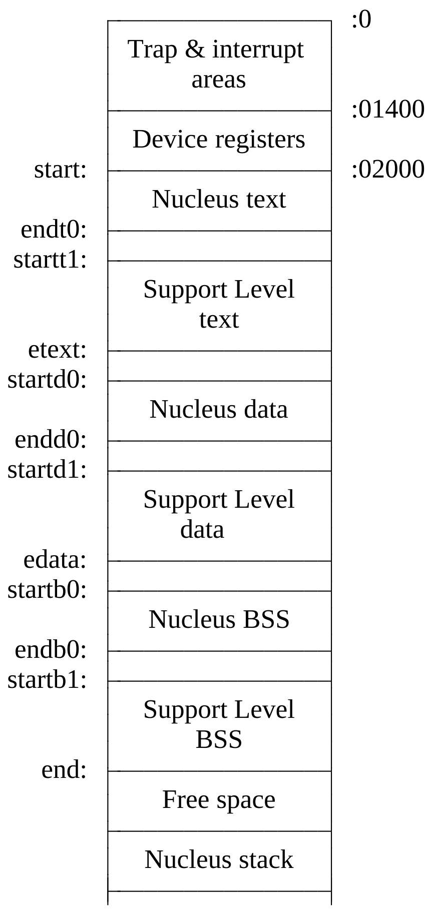

Figure 1. *CHIP Memory Map*

(3) The *bss* area. Contains the storage for the program variables with no explicit initial value (they get value 0 as a default).

Storage for these three areas are allocated in the order given above.

If you load the /stage/projects4/CHIP/hoca/lib/crt1.o file in the correct position along with your (or our) nucleus and support level, the six areas (two of each of the three types) will be loaded in memory so that there is one full page of separation between them (these areas will *not* necessarily begin on page boundaries, but are guaranteed to begin on a *different* page than the end of the previous are). This is so that you can protect the nucleus from the support level by making the areas associated with it unavailable when memory mapping is turned on (as it will be case when the support level runs). There are external labels as shown in Fig. 1 associated with the beginning and end of each of the six areas. You can refer to them symbolically in your program after having declared them as external *functions*. Refer to §6 in Chapter 5 where we discussed how to initialize a variable with the address of a function.

In Phase 3, you have the option of using your nucleus (after fixing any bugs) or using ours. To use our nucleus, link to the file /stage/projects4/CHIP/hoca/nucleus/nucleus.o in your phase3 directory and then place that file before /stage/projects4/CHIP/hoca/lib/crt1.o when loading with your support level files. If your nucleus works, using it may be to your advantage, since you will probably be able to debug more easily. The symbol table of our nucleus is in the file nucleus.symbol (same directory) for you to look at. *WARNING!* Make sure you do not have external symbol name clashes between our nucleus and your support level. (Remember that two different symbols must differ in the first 7 characters.)

2. Using Memory Mapping

The memory mapping hardware (explained in the *Documentation for the CHIP Computer System*) can be used effectively by the support level to help protect the nucleus from trusted process errors.

Consider the addresses used by the support level if memory mapping is off (that is, in the code produced by the **pcc** compiler). These addresses should be real addresses, since the support level may need to pass them to the nucleus (semaphore addresses, for example). As shown above, the nucleus and the support level will occupy the pages of memory immediately after the device registers. Viewing support level addresses as virtual addresses, they would be in segment 0 or possibly segment 1 depending on the size. Thus, if a process runs correctly under such conditions, it will still run correctly with memory mapping *on* if it uses a page table that simply maps virtual address *v* to physical address *v*. (All system processes can of course use the same segment and page tables.)

An advantage of this scheme is that the page table used by system processes and support level handlers can have the pages containing the nucleus marked as ‘‘not present.’’ Since a properly functioning process should never attempt to access a location inside the nucleus, this should cause no problems. On the other hand, if a runaway process should attempt to access the nucleus, it will generate a memory management trap and fail.

Admittedly, this scheme does nothing to prevent a malicious system process from damaging the system; however, our concern here is with accidents, not sabotage.

As the T-processes don't have segments 0 and 1 they can not address the nucleus or the support level (provided the page frames assigned to the T-processes come from the area marked as ''free space'' in Fig. 1).

You may find that the combined size of your nucleus and support level will exceed 32 pages (one segment). However, everything should fit in 64 pages (two segments). Regardless of how many pages of free memory you have after the nucleus and support level allocations, you should restrict it to 10 page frames as the test programs rely on this size to force paging to occur within a reasonable length of time.

3. T-process and Protection

Consider what happens inside the nucleus if a process running in user mode generates a trap.

Suppose a process running in user mode does indeed generate an SYS trap. The state of the process is saved (by the hardware) in the SYS old area, the new state is loaded (by the hardware), and the nucleus begins running. The nucleus finds that the process that issued the SYS was in user mode, so the SYS is treated as a software trap and, if the process has done the appropriate SYS5, is passed up. The new state will be loaded from the ''new'' area of SYS trap state vector as specified by the SYS5. We would like to have some sort of prespecified routine (the handler) to handle the trap and, for example, perform I/O to the terminal. In a similar fashion, if the process generates a legal ''missing page'' memory management trap, the handler should arrange for the page to be brought into memory.

The problem here, of course, is if the process is running in user mode, how could it have done the SYS5's in the first place? For that matter, how could we be sure that the ''new'' area specified is trustworthy? The answer to this lies in process creation. If a process does an SYS1 to create another process whose PS1 specifies user mode, then that process could never do an SYS5. However, a process running in kernel mode *can* do an LDST to go into user mode. Suppose that as part of system initialization we run a trusted kernel mode process for each terminal. This process does the SYS5's (and any other necessary housekeeping tasks) and *then* does an LDST to transfer control to the untrustworthy user program (and go into user mode). Then, a trap generated by the user code will cause control to be transferred to the location specified by the SYS5, which is presumably trustworthy. The handler will take any necessary action, then return control to the user code by doing an LDST from the SYS5 old area.

It is important to realize that there is *only one* process associated with each terminal. The nucleus remains blissfully unaware that all of these LDSTs are being done, and knows only about the original SYS1. Thus, these ''T-processes'' lead a rather schizophrenic existence; part of the time they are executing user-supplied code in user mode, while the rest of the time they are executing the trusted, system-supplied trap handler.

4. Practical Considerations

Suppose a T-process wants to read a line from its terminal. It does an SYS9, specifying a (legal) address in segment 2. The handler does the I/O, and must then move the line of input into the user's memory space. It could examine the page table to determine where this is, but a simpler solution is to give the interrupt handler access to the page tables used by the user code and do a straightforward MOVBCK to move the data to the desired location. The segment tables of the support level handlers and of the T-processes are thus the same except that the handler has access to segment 1 while the T-processes do not.

One minor subtlety here is that the handler's attempt to move the data to the user space could cause a page fault, but as long as the page fault handler gets no traps, and it shouldn't, this recursion can only be one level deep. The implication is that the support level handler for SYS and program traps will have to save the corresponding SYS5 old area somewhere else.

5. Understanding Memory in the Support Level

5.1. The Problem

When running the support level, one should realize that we are now dealing with three sets of data:

1) nucleus data
2) support level data
3) user data.

All of these data have to be located somewhere in memory and should somehow be kept secure and separate from other data. Furthermore, with regard to support level data, one should bear in mind that there will be several T-processes running, and one must ensure that each is capable of maintaining its support level data secure from those of the other T-processes. For instance, there may be two T-processes requesting to read from their respective terminals and one function which performs the terminal reads. We want both T-processes to be able to call this function concurrently without there being interference between the two with regard to the local data of that function. To see how this can be accomplished, let's look at how the C-compiler sets up our programs upon compilation and where the data is located.

5.2. Object Code

As indicated in the Fig. 1, the C-compiler generates two segments† of output: text (code) and data (for the purposes of this chapter we consider the ''data'' and ''bss'' areas together). The text segment contains all the executable instructions that constitute the

† Here we are using the word ''segment'' in a generic sense meaning a group of related locations. These ''segments'' should not be confused with the segments of the memory mapping hardware of the CHIP.

program that was compiled, and only those instructions. The data segment contains all data that are declared externally to any function. These data are static and are retained between function calls. Data that are local to specific functions do not appear in either of these segments. Space for these data is allocated dynamically *on the stack* when the corresponding function is called.

5.3. Reentrant Code

A function is considered *reentrant* if more than one process can be executing it concurrently. We wish to make the functions that make up the support level handlers reentrant so that any of the T-processes may be executing them at the same time. To do so, the data that are private to each T-process must be kept disjoint.

The data that are local to one such function will be placed on the stack when that function is called. As long as we ensure that each T-process keeps its support level stack disjoint from the others, we can be sure that there be no interference. We can enforce that each T-process has its support level stack disjoint from those of the others by the way we set up the states which are specified as trap vectors; more on that below.

There may be data that should be private to a given T-process but that are not local to one specific function. For instance, we will probably want each T-process to have one specific semaphore on which it will block when it must stop processing, e.g., when it is waiting for delay, I/O, etc. This semaphore must be accessed by several different functions, including those not in the support level handlers (for example, the clock process must be able to V this semaphore to release processes which delayed themselves). This semaphore cannot be local to any function and instead of being on the T-process's private stack it will be in the data segment as set up by the compiler. Because there will be more than one T-process running, we will have to explicitly declare more than one such semaphore. In general, we will want to declare an array of variables (indexed by T-process) for every variable we wish to be private to a T-process that is *not* local to a specific function.

5.4. Implementing Reentrant Code—Keeping Stacks Disjoint

As noted earlier, the stacks for the support level handlers will have to be disjoint between the T-processes. Recall that the different interrupt and trap handlers in the nucleus could all share the same stack (i.e., the new states loaded upon a trap or interrupt all had the same SP and STL). This was for two reasons. First, no more than one process could be executing in the nucleus at a given time, and when that process exited the nucleus it was finished with whatever it had to do (in the nucleus) at that time. Secondly, it was not possible for a process in the nucleus to generate a trap (actually program traps were possible, but these indicated a serious problem with the nucleus itself), and all interrupts were masked. Therefore, a process could not ''reenter'' the nucleus while it was already in it.

The situation with the support level is quite different. For one thing, there are several T-processes running at one time, and no assurance that when one process enters the support level, none of the others will also enter it concurrently. Therefore, each T-process needs its own support level stack. Also, a T-process executing in the support level could cause another support level trap to occur. The only way this can happen is if a process executing in the support level SYS handler causes a memory management trap (consider the case where a process tries to do a virtual-P on a semaphore that is not in physical memory). Therefore the stack used by the memory management trap handler in the support level must be disjoint from that of the SYS (and program) trap handlers.

Our root process (‘‘test’’ — the process that will actually create the T-processes) will probably execute some loop that will, with each iteration, spawn a new T-process. Each T-process will be in kernel mode as it starts up because it has to do its SYS5’s, etc. In setting up the states for the created T-processes, the root process should give each a disjoint stack. This can be done by setting the SP and STL registers in the state of the created process:

<*create start state for T-process*>;
for (each terminal) {
    <*create process*>;
    <*increment SP and STL in start state saved by some fixed amount*>;
}

It is then the responsibility of the initialization code for the T-processes to separate their stacks into two disjoint parts, one for the SYS and program trap handlers, and one for the memory management trap handler. A T-process’s stack area (as allocated by the root process) may end up looking something like this:

	←	STL at T-process startup; STL for memory management trap handler.
Memory Management stack		
	←	SP for memory management trap handler; STL for SYS and program trap handlers.
SYS & Program stack		
	←	SP at T-process startup; P for SYS and program trap handlers.

5.5. Implementing Reentrant Code—Non-local Data

As outlined above, a good solution of maintaining data private to a T-process that is not local to a specific function is to make an array indexed by the number of T-processes. If this is done, however, there is the problem of how to select the correct element of this array. That is, each T-process must somehow know its terminal number when it begins executing. One solution is to keep the terminal number in a register; we suggest R4. Remember that our C-compiler will allocate R4, R3 and R2 (*in that order*) to the first three register variables declared in any function. Therefore, we suggest that the declarations of the functions that begin the support level handlers (and the initialization function for the T-processes) begin with

```
register int    r4, r3, r2;
```

as our test programs for the nucleus often did. When creating the T-processes, the root process can ensure that each T-process starts up with its terminal number in R4. The function that begins the T-processes can initialize its trap vectors (set up with SYS5) by putting its terminal number into R4 in the new states that will be loaded up on a support level trap. In this way, any of the support level trap handlers will ‘‘come alive’’ with their terminal number in R4. After this, the terminal can either be kept in R4 (if you’re careful - remember you sometimes need R4 to pass information to the nucleus) or passed as a parameter to other functions that may be part of the support level handlers.

5.6. Shared Data

There will, of course, be data that must be shared between the different T-processes. For instance, the ‘‘Delay’’ SYS call probably places the invoking T-process on a queue which the clock process will periodically examine. We need to be able to maintain the integrity of data structures such as this queue. To do so we can use a very simple solution: use semaphores implemented by the nucleus to guard entry to critical sections. With every shared data structure we will associate a semaphore which will control access to any operations on the data structure.

5.7. User Data in the Test Programs

Each of the test programs has its own data, some of which is private to the T-process. However, there is no reason to worry about these private data of the T-processes. They will be in segment two (private), and the virtual memory system will keep them for conflicting with those of the other T-processes.

Appendix One
Module Example

```
/*******************************QUEUE.C***************************************
 *
 *      Module: FIFO queue of positive integers of maximum size QSIZE.
 *      Invariant: 0 <= count,next_out <= QSIZE and
 *              queue[next_out ... (next_out + count - 1) % QSIZE] is full
 *
 *      Implemented as a circular buffer in the integer array queue[QSIZE].
 *      Count is the current number of elements in the queue.
 ****************************************************************************/

#include "defs.h"
#define QSIZE       20

HIDDEN int count, next_out;
HIDDEN int queue[QSIZE];

/* Initialize empty queue. */
initq()
{
        count = next_out = 0;
}

/*
 * Insert an element into the queue if not full.
 * Otherwise, return error.
 */
int
insertq(e)
        register int e;
{
        if (count == QSIZE)
                return(ERROR);
        else {
                queue[(next_out + count) % QSIZE] = e;
                count++;
                return(e);
        }
}

/*
 * Remove an element from the queue.  If empty,
 * return error.  Otherwise, return element.
 */
int
```

```
removeq()
{
	register int e;

	if (count == 0)
		return(ERROR);
	else {
		e = queue[next_out];
		next_out = (next_out + 1) % QSIZE;
		count--;
		return(e);
	}
}
```

```
/******************************* QUEUE.E ***************************************/
extern int insertq();
extern int removeq();
extern int initq();
```

```
/******************************* DEFS.H ***************************************/
#define TRUE 1
#define FALSE	0
#define ERROR	-1
#define HIDDEN	static
```

```
/******************************* PROG.C **************************************/
#include "defs.h"
#include "queue.e"
	...
main()
{
	int x;
	...
	initq();
	...
	if (insertq(x) == ERROR)
		printf("error: queue empty0);
		...
	if ((x=removeq()) == ERROR) {
```

```
                printf("error: queue full0);
                return;
        } else {
                ...
        }
}
```

Appendix Two

CHIP Hardware Type Definitions

```
/* Hardware constants */
#define PAGESIZE       512       /* page size in bytes */
#define BEGINTRAP      0         /* beginning of trap areas */
#define BEGININTR      0220      /* beginning of interrupt areas */
#define BEGINDEVREG    01400     /* beginning of device registers */

/* utility constants */
#define TRUE           1
#define FALSE          0
#define NULL           0177777
#define NOPROC         -1
#define SECOND         1000000L

/* trap types */
#define PROGTRAP       0         /* program trap */
#define MMTRAP         1         /* memory management trap */
#define SYSTRAP        2         /* system call trap */
#define TRAPTYPES      3         /* number of trap types */

/* program trap codes */
#define ILOPCODE       0         /* illegal opcode */
#define NONSUPINSTR    1         /* nonsupported instruction */
#define PRIVINSTR      2         /* privileged instruction */
#define ILLINSTR       3         /* ill formed instruction */
#define ODDADD         4         /* odd address */
#define NONPAGEAL      5         /* non page alignment */
#define NONEXISTMEM    6         /* non existent memory */
#define STKLIMYELLOW   7         /* stack limit yellow */
#define STKLIMRED      8         /* stack limit red */
#define ZERODIV        9         /* zero divide */

/* memory management trap codes */
#define ACCESSPROT     0         /* access protection violation */
#define PAGEMISS       1         /* missing page */
#define INVPAGE        2         /* invalid page number */
#define SEGMISS        3         /* missing segment */

/* device operation codes */
#define IOREAD         0
#define IOWRITE        1
#define IOSEEK         2

/* device completion codes */
#define NORMAL         0
```

```
#define HARDFAILURE     1
#define ILOP            2
#define ILBUF           3
#define ILAMOUNT        4
#define ILTRACK         5
#define ILSECTOR        6
#define ENDOFINPUT      7
#define NOSUCHDEV       8
#define DEVNOTREADY     9

/* device numbers according to the position of their device registers */
#define TERM0           0
#define TERM1           1
#define TERM2           2
#define TERM3           3
#define TERM4           4
#define PRINT0          5
#define PRINT1          6
#define DISK0           7
#define DISK1           8
#define DISK2           9
#define DISK3           10
#define DRUM0           11
#define DRUM1           12
#define DRUM2           13
#define DRUM3           14

/* operations */
#define MIN(A,B)        ((A) < (B) ? A : B)
#define MAX(A,B)        ((A) < (B) ? B : A)
#define EVEN(A)         (((unsigned)A & 01) == 0)
```

Constants

```
/* virtual address */
typedef struct {
	unsigned v_offset :9,		/* byte offset in page */
		v_page   :5,		/* page number */
		v_seg    :2;		/* segment number */
} vad_t;

/* page descriptor */
typedef struct {
	unsigned pd_frame :8,		/* page frame number */
			 :5,		/* unused */
		pd_r     :1,		/* reference bit */
		pd_m     :1,		/* modified bit */
		pd_p     :1;		/* presence bit */
} pd_t;

/* segment descriptor */
typedef struct {
	unsigned sd_len		:5,		/* page table length */
		sd_prot :3,		/* access protection bits */
		sd_p    :1,		/* presence bit */
			:7;		/* unused */
	pd_t	*sd_pta;		/* address of the page table */
} sd_t;

/* processor status word1 */
typedef struct {
	unsigned ps_c :1,		/* carry */
		ps_o  :1,		/* overflow */
		ps_z  :1,		/* zero */
		ps_n  :1,		/* negative */
		      :1,		/* not used */
		ps_w  :1,		/* wait */
		ps_m  :1,		/* translation */
		ps_ku :1,		/* processor mode */
		ps_int :8;		/* interrupt mask */
} ps1_t;

/* processor status word2 */
typedef union {
	struct {
		unsigned in_dno		:3,		/* device number */
			in_dev:3,		/* device type */
				:10;		/* not used */
	} p2_int;
	struct {
```

```
		unsigned mm_pg		:5,		/* page number */
			 mm_seg		:2,		/* segment number */
				:1,		/* not used */
			 mm_typ :8;		/* type of mm trap */
	} p2_mm;
	struct {
		unsigned pr_typ		:8,		/* type of prog trap */
				:8;		/* not used */
	} p2_pr;
	struct {
		unsigned sys_no		:8,		/* number of sys trap */
				:8;		/* not used */
	} p2_sys;
} ps2_t;

/* processor state */
typedef struct {
	int	s_r[8];		/* general registers r0-r5 */
#define s_sp	s_r[6]			/* stack pointer */
#define s_pc	s_r[7]			/* program counter */
	ps1_t	s_ps1;			/* processor status word 1 */
	ps2_t	s_ps2;			/* processor status word 2 */
	sd_t	*s_sta;			/* segment table address register */
	unsigned s_stl;			/* stack limit register */
} state_t;

/* Device registers */
typedef		struct {
	unsigned d_op,			/* operation register */
		 d_dadd;	/* address, amount, track or sector number */
#define d_amnt		d_dadd		/* synonyms for the above */
#define d_track		d_dadd
#define d_sect	d_dadd
	char	*d_badd;	/* buffer address register */
	unsigned d_stat,	/* status register */
		 d_notused[4];	/* force alignment at each 8th word */
} devreg_t;


typedef		union {
	long	l;
	struct	{
		unsigned int	whigh,
				wlow;
	} w;
} double_t;
```

```
typedef         struct {
        int     io_len;
        int     io_sta;
} iores_t;
```