The HOCA Operating System Specifications

Özalp Babaoğlu
Fred B. Schneider

Department of Computer Science
Cornell University
Ithaca, New York 14853

Revised for use at
Department of Math and Computer Science
Emory University
Atlanta, Georgia 30322

K. Mandelberg
E. Leon

January 10, 2022

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. Late Cornell versions of the project were designed to be implented in C and execute on the PDP-11 simulator CHIP.

The current Emory University version of HOCA is designed to be implemented in C and executes on the EMACSIM (Emory MACintosh SIMulator) host computer. The EMACSIM architecture is based loosely on the Apple Macintosh II computer. The simulation can be run portably on many UNIX† systems, but a special peripheral interface is available when run on a Macintosh A/UX†† system.

Edgar Leon implemented the EMACSIM simulator and diagnostic interface under the direction of K. Mandelberg. Much of the present work is based on the CHIP version as well as the GDB debugger from the GNU project.

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

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

†† A/UX and Macintosh are Trademarks of Apple Computer.

cs452/cs580 Guidelines

In this course you will write an operating system which consists of three phases: queues, nucleus and support. The nucleus and support phases have two parts each. Your grade will be computed in the following way:

Undergraduate

PHASE	%	DUE DATE	COMMENTS
queues	15%	Monday, Feb. 7	
nucleus-part a	20%	Monday, Feb. 28	
nucleus-part b	20%	Monday, March 21	
support-part a	20%	Monday, April 25	last day of classes
final	25%		

Graduate

PHASE	%	DUE DATE	COMMENTS
queues	11%	Monday, Feb. 7	
nucleus-part a	16%	Monday, Feb. 28	
nucleus-part b	16%	Monday, March 21	
support-part a	16%	Thursday, April 6	
support-part b	16%	Monday, April 25	last day of classes
final	25%		

You should create four directories under your cs452/cs580 directory, and your programs should be copied to the corresponding directory. The programs are due according to the schedule listed above. Note that there will be a late penalty, 10% of your grade will be lost for every business day that your program is late. Programs can only be "handed in" on business days.

Since 75% of the grade in this course is based on the OS project, it is essential that code you hand is YOUR OWN WORK. It is an HONOR COUNCIL violation to hand in code done in consultation with other students in or out of the course, or consult solutions done by another student in a past semester. While general discussion of algorithms or requirements is allowed and encouraged, your code must be your own.

Each module should include the following statement:

This code is my own work, it was written without consulting code written by other students (My Name)

INSTRUCTOR

Name:	Dr. Ken Mandelberg	
Email:	km@emory.edu	
Office:	Online	
Office Hours:	Tuesdays	4:00 pm - 5:30 pm
	Thursdays	4:00 pm - 5:30 pm
	and also by appointment.	

GRADER

Name:	Yuting Guo
Email:	yuting.guo@emory.edu
Office:	Online
Office Hours:	TBD
	and also by appointment.

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 Emory, the HOCA implementation project accounts for the laboratory component of a beginning graduate level course in Operating Systems.

The host machine for the project is a Macintosh II like computer computer called EMACSIM (Emory MACintosh SIMulator). A description of the EMACSIM architecture is provided separately.

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 EMACSIM 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
File Organization

HOCA is implemented as a set of C programming language modules.
A module is made up of three different types of files, with extensions as follows:

".h" - Header files
".e" - Export files
".c" - *C code*

For each module **X**, you should create three files: **X**.h, **X**.e and **X**.c.
X.e and **X**.c should include **X**.h.

Header files contain constants, macros and types.

Export files contain constants, macros, types, variables and function prototypes that are the module definitions visible to other modules, For example, Module1 might consist of mod1.c, mod1.h and mod1.e. If Module2 needs to use the services of Module1, then mod2.c would include mod1.e. If Module2 includes multiple export files it must do so in a way that will avoid the same header file being included multiple times.

Constants, macros, types and variables that are local to module **X** and that are not used by any other module should be declared at the top of the file **X**.c.

Examples.

Header file example: **head.h**

```
#define TRUE 1
#define MAXPROC 20
typedef struct vars {
   int i;
   int *j;
} vars;
```

Export file example: **exp.e**

```
#include "head.h"
extern int function1();
extern vars *function2();
```

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

You will use the **#include** mechanism of the C preprocessor to include the header and export files that are being imported.

A detailed example of a FIFO queue can be found in Appendix 1.

You should adhere to these file organization guidelines.

Chapter Three
Program Guidelines

1. Program Structure

(a) Global structure: The file structure should conform to the description given in Chapter 2. Namely, each ‘‘.c’’ file should have an associated ‘‘.e’’ file if there are any declarations visible to other programs.

(b) Indentation: 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.

(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.

2. Layout of Comments

Every ‘‘.c’’ file should have a header comment as follows:

(a) Honor Code Statement.

(b) A brief description of what the program does.

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 */
            ...
}
```

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.

3. 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: The Queues
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.

You are strongly encouraged to create four subdirectories in your class directory: *queues, nucleus, support and h*. Three of them will contain the code for each of the three phases, and *h*, will contain the included (i.e., ".e" and ".h") files. The *nucleus* and *support* directories will be subdivided later. Appendix 2 contains the listing of two files defining certain hardware-related constants and types for the EMACSIM computer. These will be very useful for you. The two files are available in the directory ˜cs580000/share/hoca/h under the names *types.h* and *const.h*. You should copy them into the *h* subdirectory of your account and you should not change these files. For this assignment, you will also need the following type definitions.

```
/* link descriptor type */
typedef struct proc_link {
      int index;
      struct proc_t *next;
} proc_link;

/* process table entry type */
typedef struct proc_t {
      proc_link  p_link[SEMMAX];     /* pointers to next entries */
      state_t        p_s;            /* processor state of the process */
      int qcount;                    /* number of queues containing this entry */
      int *semvec[SEMMAX];           /* vector of active semaphores for this entry */
      /* 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_link         s_link;         /* pointer/index 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. Each function definition should be preceded by a precise description of *what* it does and not *how* it is implemented.

You must 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 ˜cs580000/share/hoca/queues and do not modify this file. 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 ENULL-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.

The module should have the following externally visible functions. Note, since a proct_t can reside on more than one queue at a time, all functions are responsible for properly updating the qcount and proc_link fields to reflect the operation performed.

If your function finds an error condition that it cannot handle or recover from, it should panic (stop the simulator). If an error condition is found, please copy an error message to an error buffer and issue this command: asm(" trap #0"); Please write a panic function similar to the adderrbuf function in the test program.

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

Insert the element pointed to by *p* into the process queue where *tp* contains the pointer/index to the tail (last element). Update the tail pointer accordingly. If the process is already in SEMMAX queues, call the panic function.

proc_t *
removeProc(tp)
    proc_link *tp;

Remove the first element from the process queue whose tail is pointed to by *tp*. Return ENULL 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_link *tp;
    proc_t *p;

> Remove the process table entry pointed to by *p* from the queue whose tail 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 ENULL. Otherwise, return *p*. Note that *p* can point to any element of the queue (not necessarily the head element).

proc_t *
allocProc()

> Return ENULL 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.

proc_t *
headQueue(tp)
    proc_link tp;

> Return a pointer to the process table entry at the head of the queue. The tail of the queue is pointed to by *tp*.

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 ENULL-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. Note, since a proct_t can reside on more than one queue at a time, all functions are responsible for properly updating the semvec fields to reflect the operation performed.

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 ENULL. 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 queues 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 ENULL. 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 ENULL.

initSemd()

Initialize the semaphore descriptor free list.

headASL()

This function will be used to determine if there are any semaphores on the ASL. Return FALSE if the ASL is empty or TRUE if not empty.

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.

Header and Export File Examples.

Header file example: **h/procq.h**

```
/* link descriptor type */
typedef struct proc_link {
    int index;
    struct proc_t *next;
} proc_link;

/* process table entry type */
typedef struct proc_t {
    proc_link  p_link[SEMMAX];     /* pointers to next entries */
    state_t       p_s;          /* processor state of the process */
    ...
    /* plus other entries to be added by you later */
} proc_t;
```

Export file example: **h/procq.e**

```
#include "procq.h"
extern int insertProc();
extern proc_t *removeProc();
 ...
/* plus other entries to be added by you later */
```

C Program Examples.

Program name: **queues/procq.c**

```
HONOR-CODE-STATEMENT
BRIEF-DESCRIPTION
#include "../h/const.h"
#include "../h/types.h"
#include "../h/procq.h"
 ...
```

Program name: **queues/asl.c**

```
HONOR-CODE-STATEMENT
BRIEF-DESCRIPTION
#include "../h/types.h"
#include "../h/const.h"
#include "../h/procq.e"
#include "../h/asl.h"
 ...
```