Appendix O
BACI: The Ben-Ari Concurrent Programming System

By Bill Bynum, College of William and Mary
Tracy Camp, Colorado School of Mines

O.1 INTRODUCTION
O.2 BACI
System Overview
Concurrency Constructs in BACI
How to Obtain BACI
O.3 EXAMPLES OF BACI PROGRAMS
O.4 BACI PROJECTS
Implementation of Synchronization Primitives
Semaphores, Monitors, and Implementations
O.5 ENHANCEMENTS TO THE BACI SYSTEM

Supplement to
Operating Systems, Eighth Edition
Pearson 2014

O.1 INTRODUCTION

In Chapter 5, concurrency concepts are introduced (e.g., mutual exclusion and the critical section problem) and synchronization techniques are proposed (e.g., semaphores, monitors, and message passing). Deadlock and starvation issues for concurrent programs are discussed in Chapter 6. Due to the increasing emphasis on parallel and distributed computing, understanding concurrency and synchronization is more necessary than ever. To obtain a thorough understanding of these concepts, practical experience writing concurrent programs is needed.

Three options exist for this desired “hands-on" experience. First, we can write concurrent programs with an established concurrent programming language such as Concurrent Pascal, Modula, Ada, or the SR Programming Language. To experiment with a variety of synchronization techniques, however, we must learn the syntax of many concurrent programming languages. Second, we can write concurrent programs using system calls in an operating system such as UNIX. It is easy, however, to be distracted from the goal of understanding concurrent programming by the details and peculiarities of a particular operating system (e.g., details of the semaphore system calls in UNIX). Lastly, we can write concurrent programs with a language developed specifically for giving experience with concurrency concepts such as the Ben-Ari Concurrent Interpreter (BACI) [BYNU96]. Using such a language offers a variety of synchronization techniques with a syntax that is usually familiar. Languages developed specifically for giving experience with concurrency concepts are the best option to obtain the desired hands-on experience.

Section O.2 contains a brief overview of the BACI system and how to obtain the system. Section O.3 contains examples of BACI programs, and Section O.4 contains a discussion of projects for practical concurrency experience at the implementation and programming levels. Lastly, Section O.5 contains a description of enhancements to the BACI system that have been made.

O.2 BACI

System Overview

BACI is a direct descendant of Ben-Ari's modification to sequential Pascal (Pascal-S). Pascal-S is a subset of standard Pascal by Wirth, without files, except INPUT and OUTPUT, sets, pointer variables, and goto statements. Ben-Ari took the Pascal-S language and added concurrent programming constructs such as the `cobegin ... coend` construct and the semaphore variable type with `wait` and `signal` operations [BEN82]. BACI is Ben-Ari's modification to Pascal-S with additional synchronization features (e.g., monitors) as well as encapsulation mechanisms to ensure that a user is prevented from modifying a variable inappropriately (e.g., a semaphore variable should only be modified by semaphore functions).

BACI simulates concurrent process execution and supports the following synchronization techniques: general semaphores, binary semaphores, and monitors. The BACI system is composed of two subsystems, as illustrated in Figure O.1. The first subsystem, the compiler, compiles a user's program into intermediate object code, called PCODE. There are two compilers available with the BACI system, corresponding to two popular languages taught in introductory programming courses. The syntax of one compiler is similar to standard Pascal; BACI programs that use the Pascal syntax are denoted as pgrm-name.pm. The syntax of the other compiler is similar to standard C++; these BACI programs are denoted as pgrm-name.cm. Both compilers create two files during the compilation: pgrm-name.lst and pgrm-name.pco.

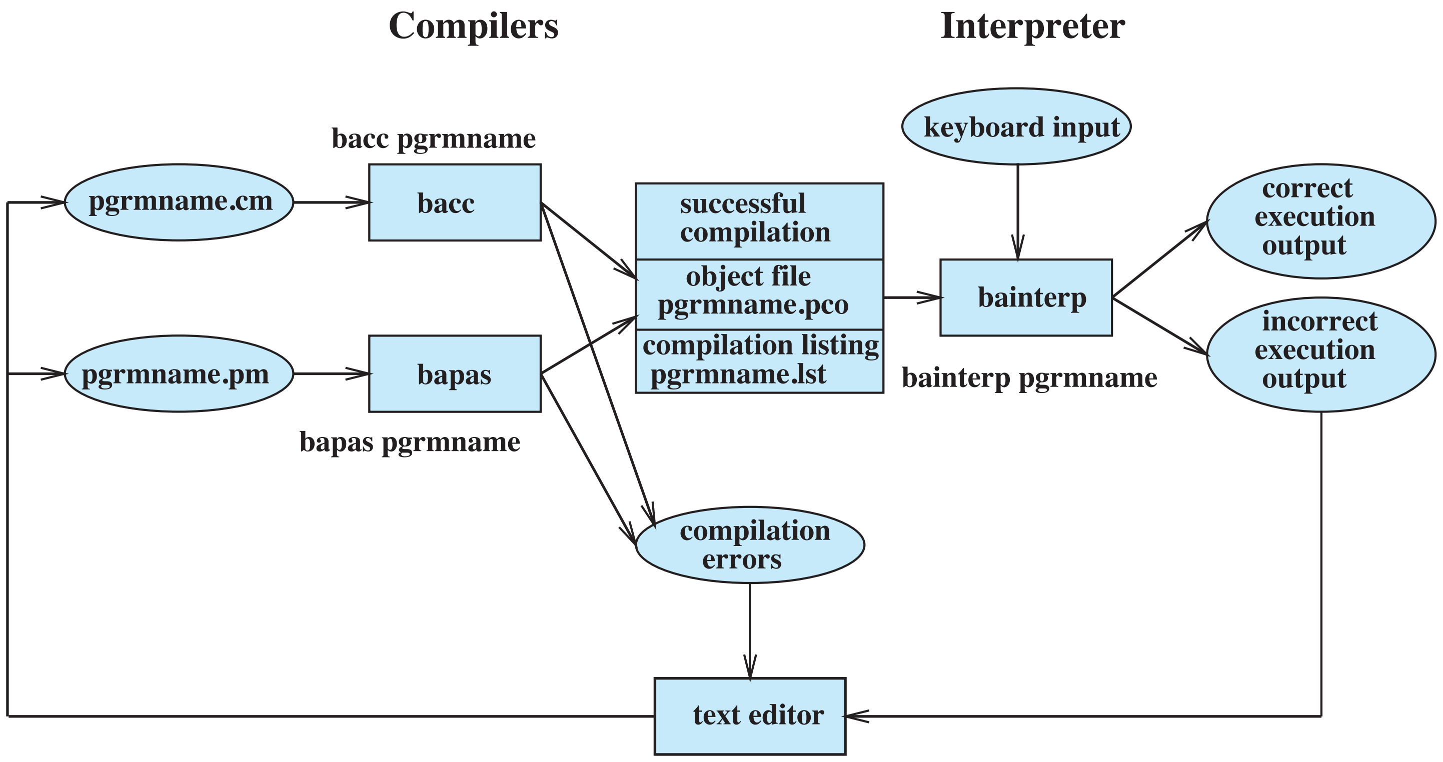

Figure O.1 Overview of the BACI System

The second subsystem in the BACI system, the interpreter, executes the object code created by the compiler. In other words, the interpreter executes pgrm-name.pco. The core of the interpreter is a preemptive scheduler; during execution, this scheduler randomly swaps between concurrent processes, thus simulating a parallel execution of the concurrent processes. The interpreter offers a number of different debug options, such as single-step execution, disassembly of PCODE instructions, and display of program storage locations.

Concurrency Constructs in BACI

In the rest of this appendix, we focus on the compiler similar to standard C++. We call this compiler C--; although the syntax is similar to C++, it does not include inheritance, encapsulation, or other object-oriented programming features. In this section, we give an overview of the BACI concurrency constructs; see the user's guides at the BACI Web site for further details of the required Pascal or C-- BACI syntax.

COBEGIN

A list of processes to be run concurrently is enclosed in a cobegin block. Such blocks cannot be nested and must appear in the main program.

```
cobegin { proc1(...); proc2(...); ... ; procN(...); }
```

The PCODE statements created by the compiler for the above block are interleaved by the interpreter in an arbitrary, “random” order; multiple executions of the same program containing a cobegin block will appear to be nondeterministic.

SEMAPHORES

A semaphore in BACI is a nonnegative-valued `int` variable, which can only be accessed by the semaphore calls defined subsequently. A binary semaphore in BACI, one that only assumes the values 0 and 1, is supported by the `binarysem` subtype of the `semaphore` type. During compilation and execution, the compiler and interpreter enforce the restrictions that a `binarysem` variable can only have the values 0 or 1 and that `semaphore` type can only be nonnegative. BACI semaphore calls include

- `initialsem(semaphore sem, int expression)`
- `p(semaphore sem)`: If the value of `sem` is greater than zero, then the interpreter decrements `sem` by one and returns, allowing `p`'s caller to continue. If the value of `sem` is equal to zero, then the interpreter puts `p`'s caller to sleep. The command `wait` is accepted as a synonym for `p`.
- `v(semaphore sem)`: If the value of `sem` is equal to zero and one or more processes are sleeping on `sem`, then wake up one of these processes. If no processes are waiting on `sem`, then increment `sem` by one. In any event, `v`'s caller is allowed to continue. (BACI conforms to Dijkstra's original semaphore proposal by randomly choosing which process to wake up when a signal arrives.) The command `signal` is accepted as a synonym for `v`.

MONITORS

BACI supports the monitor concept, as proposed by Hoare [HOAR74], with some restrictions; the implementation is based on work done by [PRAM84]. A monitor is a C++ block, like a block defined by a procedure or function, with some additional properties (e.g., conditional variables). In BACI, a monitor must be declared at the outermost, global level and it cannot be nested with another monitor block. Three constructs are used by the procedures and functions of a monitor to control concurrency: condition variables, `waitc` (wait on a condition), and `signalc` (signal a condition). A condition never actually has a value; it is somewhere to wait or something to signal. A monitor process can wait for a condition to hold or signal that a given condition now holds through the `waitc` and `signalc` calls. `waitc` and `signalc` calls have the following syntax and semantics:

- `waitc(condition cond, int prio)`: The monitor process (and hence the outside process calling the monitor process) is blocked on the condition `cond` and assigned the priority `prio`.
- `waitc(condition cond)`: This call has the same semantics as the `waitc` call, but the `wait` is assigned a default priority of 10.
- `signalc(condition cond)`: Wake some process waiting on `cond` with the smallest (highest) priority; if no process is waiting on `cond`, do nothing.

BACI conforms to the immediate resumption requirement. In other words, a process waiting on a condition has priority over a process trying to enter the monitor, if the process waiting on a condition has been signaled.

OTHER CONCURRENCY CONSTRUCTS

The C-- BACI compiler provides several low-level concurrency constructs that can be used to create new concurrency control primitives. If a function is defined as atomic, then the function is nonpreemptible. In other words, the interpreter will not interrupt an atomic function with a context switch. In BACI, the suspend function puts the calling process to sleep and the revive function revives a suspended process.

How to Obtain BACI

The BACI system, with two user guides (one for each of the two compilers) and detailed project descriptions, is available at the BACI Web site (there is a link to this Web site from WilliamStallings.com/OS/OS6e.html). The BACI system is written in both C and Java. The C version of the BACI system can be compiled in Linux, RS/6000 AIX, Sun OS, DOS, and CYGWIN on Windows with minimal modifications to the Makefile file. (See the README file in the distribution for installation details for a given platform.) The BACI system is in use (or has been used) at more than 60 universities in 25 countries (as of September 2007). Please email bynum@cs.wm.edu or tcamp@mines.edu if you are using BACI at your university.

O.3 EXAMPLES OF BACI PROGRAMS

In Chapters 5 and 6, a number of the classical synchronization problems were discussed (e.g., the readers/writers problem and the dining philosophers problem). In this section, we illustrate the BACI system with three BACI programs. Our first example illustrates the nondeterminism in the execution of concurrent processes in the BACI system. Consider the following program:

```
const int m = 5;
int n;
void incr(char id)
{
     int i;
     for(i = 1; i <= m; i = i + 1)
      {
        n = n + 1;
        cout << id << " n =" << n << " i =";
        cout << i << " " << id << endl;
      }
}
main()
{
     n = 0;
     cobegin {
       incr( 'A'); incr( 'B' ); incr('C');
      }
```

```
    cout << "The sum is " << n << endl;
}
```

Note in the preceding program that if each of the three processes created (A, B, and C) executed sequentially, the output sum would be 15. Concurrent execution of the statement n = n + 1;, however, can lead to different values of the output sum. After we compiled the preceding program with bacc, we executed the PCODE file with bainterp a number of times. Each execution produced output sums between 9 and 15. One sample execution produced by the BACI interpreter is the following.

```
Source file: incremen.cm Fri Aug 1 16:51:00 1997
CB n =2 i =1 C n =2
A n =2 i =1 i =1 A
CB
    n =3 i =2 C
A n =4 i =2 C n =5 i =3 C
A
B n =6C i =2 B
    n =7 i =4 C
A n =8 i =3 A
BC n =10 n =10 i =5 C
A n = i =311 i =4 A
 B
A n =12 i =B5  n =13A
    i =4 B
B n =14 i =5 B
The sum is 14
```

Special machine instructions are needed to synchronize the access of processes to a common main memory. Mutual exclusion protocols, or synchronization primitives, are then built on top of these special instructions. In BACI, the interpreter will not interrupt a function defined as atomic with a context switch. This feature allows users to implement these low-level special machine instructions. For example, the following program is a BACI implementation of the testset function. A testset instruction tests the value of the function’s argument i. If the value of i is zero, the function replaces it with 1 and returns true; otherwise, the function does not change the value of i and returns false. As discussed in Section 5.2, special machine instructions (e.g., testset) allow more than one action to occur without interruption. BACI has an atomic keyword defined for this purpose.

```
// Test and set instruction
//
atomic int testset(int& i)
{
     if (i == 0) {
       i = 1;
       return 1;
     }
     else
       return 0;
}
```

We can use testset to implement mutual exclusion protocols, as shown in the following program. This program is a BACI implementation of a mutual exclusion program based on the test and set instruction. The program assumes three concurrent processes; each process requests mutual exclusion 10 times.

```
int bolt = 0;
const int RepeatCount = 10;
void proc(int id)
{
     int i = 0;
     while(i < RepeatCount) {
       while (testset(bolt)); // wait
        // enter critical section
        cout << id;
       // leave critical section
       bolt = 0;
       i++;
     }
}
main()
{
     cobegin {
       proc(0); proc(1); proc(2);
     }
}
```

The following two programs are a BACI solution to the bounded-buffer producer/consumer problem with semaphores (see Figure 5.13). In this example, we have two producers, three consumers, and a buffer size of five.

We first list the program details for this problem. We then list the include file that defines the bounded buffer implementation.

```
// A solution to the bounded-buffer producer/consumer problem
// Stallings, Figure 5.13
// bring in the bounded buffer machinery
#include "boundedbuff.inc"
const int ValueRange = 20; // integers in 0..19 will be produced
semaphore to;   // for exclusive access to terminal output
semaphore s;   // mutual exclusion for the buffer
semaphore n;   // # consumable items in the buffer
semaphore e;   // # empty spaces in the buffer
int produce(char id)
{
     int tmp;
     tmp = random(ValueRange);
     wait(to);
     cout << "Producer " << id << " produces " << tmp << endl;
     signal(to);
     return tmp;
}
void consume(char id, int i)
{
     wait(to);
     cout << "Consumer " << id << " consumes " << i << endl;
     signal(to);
}
void producer(char id)
{
     int i;
     for (;;) {
       i = produce(id);
       wait(e);
       wait(s);
       append(i);
       signal(s);
       signal(n);
     }
}
void consumer(char id)
{
     int i;
     for (;;) {
       wait(n);
       wait(s);
       i = take();
       signal(s);
       signal(e);
       consume(id,i);
     }
}
main()
{
     initialsem(s,1);
     initialsem(n,0);
     initialsem(e,SizeOfBuffer);
     initialsem(to,1);
     cobegin {
       producer('A'); producer('B');
       consumer('x'); consumer('y'); consumer('z');
     }
}

// boundedbuff.inc -- bounded buffer include file
const int SizeOfBuffer = 5;
int buffer[SizeOfBuffer];
int in = 0;    // index of buffer to use for next append
int out = 0;   // index of buffer to use for next take
void append(int v)
     // add v to the buffer
     // overrun is assumed to be taken care of
     // externally through semaphores or conditions
{
     buffer[in] = v;
     in = (in + 1) % SizeOfBuffer;
}
int take()
     // return an item from the buffer
 // underrun is assumed to be taken care of
 // externally through a semaphore or condition
{
     int tmp;
     tmp = buffer[out];
     out = (out + 1) % SizeOfBuffer;
     return tmp;
}
```

One sample execution of the preceding bounded-buffer solution in BACI is the following.

```
Source file: semprodcons.cm Fri Aug 1 12:36:55 1997
Producer B produces 4
Producer A produces 13
Producer B produces 12
Producer A produces 4
Producer B produces 17
Consumer x consumes 4
Consumer y consumes 13
Producer A produces 16
Producer B produces 11
Consumer z consumes 12
Consumer x consumes 4
Consumer y consumes 17
Producer B produces 6
...
```

O.4 BACI PROJECTS

In this section, we discuss two general types of projects one can implement in BACI. We first discuss projects that involve the implementation of low-level operations (e.g., special machine instructions that are used to synchronize the access of processes to a common main memory). We then discuss projects that are built on top of these low-level operations (e.g. classical synchronization problems). For more information on these projects see [BYNU96] and the project descriptions included in the BACI distribution. For solutions to some of these projects, teachers should contact the authors. In addition to the projects discussed in this section, many of the problems at the end of Chapter 5 and Appendix A can be implemented in BACI.

Implementation of Synchronization Primitives

Implementation of Machine Instructions

There are numerous machine instructions that one can implement in BACI. For example, one can implement the compare-and-swap or the exchange instruction discussed in Figure 5.2. The implementation of these instructions should be based on an atomic function that returns an int value. You can test your implementation of the machine instruction by building a mutual exclusion protocol on top of your low-level operation.

Implementation of Fair Semaphores (FIFO)

The semaphore operation in BACI is implemented with a random wake up order, which is how semaphores were originally defined by Dijkstra. As discussed in Section 5.3, however, the fairest policy is FIFO. One can implement semaphores with this FIFO wake up order in BACI. At least the following four procedures should be defined in the implementation:

- `CreateSemaphores()` to initialize the program code
- `InitSemaphore(int sem-index)` to initialize the semaphore represented by `sem-index`
- `FIFOP(int sem-index)`
- `FIFOV(int sem-index)`

This code needs to be written as a system implementation and, as such, should handle all possible errors. In other words, the semaphore designer is responsible for producing code that is robust in the presence of ignorant, stupid, or even malicious use by the user community.

Semaphores, Monitors, and Implementations

There are many classical concurrent programming problems: the producer/consumer problem, the dining philosophers, the reader/writer problem with different priorities, the sleeping barber problem, and the cigarette smoker's problem. All of these problems can be implemented in BACI. In this section, we discuss nonstandard semaphore/monitor projects that one can implement in BACI to further aid the understanding of concurrency and synchronization concepts.

A'S AND B'S AND SEMAPHORES

For the following program outline in BACI,

```
// global semaphore declarations here

void A()
{
    p()'s and v()'s ONLY
}

void B()
{
    p()'s and v()'s ONLY
}
```

```
main()
{
     // semaphore initialization here
     cobegin {
       A(); A(); A(); B(); B();
     }
}
```

complete the program using the least number of general semaphores, such that the processes ALWAYS terminate in the order A (any copy), B (any copy), A (any copy), A, B. Use the -t option of the interpreter to display process termination. (Many variations of this project exist. For example, have four concurrent processes terminate in the order ABAA or eight concurrent processes terminate in the order AABABABB.)

USING BINARY SEMAPHORES

Repeat the previous project using binary semaphores. Evaluate why assignment and IF-THEN-ELSE statements are necessary in this solution, although they were not necessary in solutions to the previous project. In other words, explain why you cannot use only Ps and Vs in this case.

BUSY WAITING VERSUS SEMAPHORES

Compare the performance of a solution to mutual exclusion that uses busy waiting (e.g. the testset instruction) to a solution that uses semaphores. For example, compare a semaphore solution and a testset solution to the ABAAB project discussed previously. In each case, use a large number of executions (say, 1000) to obtain better statistics. Discuss your results, explaining why one implementation is preferred over another.

SEMAPHORES AND MONITORS

In the spirit of Problem 5.17, implement a monitor using general semaphores and then implement a general semaphore using a monitor in BACI.

General and Binary Semaphores

Prove that general semaphores and binary semaphores are equally powerful, by implementing one type of semaphore with the other type of semaphore and vice versa.

Time Ticks: A Monitor Project

Similar to Problem 7.17 in [SILB02], write a program containing a monitor AlarmClock. The monitor must have an `int` variable `theClock` (initialized to zero) and two functions:

- `Tick()`: This function increments `theClock` each time that it is called. It can do other things, like `signalc`, if needed.
- `int Alarm(int id, int delta)`: This function blocks the caller with identifier `id` for at least `delta` ticks of `theClock`.

The main program should have two functions as well:

- `void Ticker()`: This procedure calls `Tick()` in a repeat-forever loop.
- `void Thread(int id, int myDelta)`: This function calls `Alarm` in a repeat-forever loop.

You may endow the monitor with any other variables that it needs. The monitor should be able to accommodate up to five simultaneous alarms.

A Problem of a Popular Baker

Due to the recent popularity of a bakery, almost every customer needs to wait for service. To maintain service, the baker wants to install a ticket system that will ensure that customers are served in turn. Construct a BACI implementation of this ticket system.

O.5 ENHANCEMENTS TO THE BACI SYSTEM

We have enhanced the BACI System in several ways.

1. We have implemented the BACI system in Java (JavaBACI). It, along with our original C implementation of BACI, is available from: http://inside.mines.edu/fs_home/tcamp/baci/baci_index.html . The JavaBACI classes and source files are stored in self-extracting Java jar files. JavaBACI includes all BACI applications: C and Pascal compilers, disassembler, archiver, linker, and command-line and GUI PCODE interpreters. The input, behavior, and output of programs in JavaBACI are identical to the input, behavior, and output of programs in our C implementation of BACI; we note that, in JavaBACI, students continue to write concurrency programs in C-- or Pascal (not Java). JavaBACI will execute on any computer that has an installation of the Java Virtual Machine.
2. We have added graphical user interfaces (GUIs) for JavaBACI and the UNIX version of BACI in C. The windowing environments of these GUIs allow a user to monitor all aspects of the execution of a BACI program; specifically, a user can set and remove breakpoints (either by PCODE address or source line), view variables values, runtime stacks, and process tables, and examine interleaved PCODE execution. The BACI GUIs are available at the BACI GUI Web site

http://inside.mines.edu/fs_home/tcamp/baci/index_gui.html. For an alternative GUI, see below.

3. We have created a distributed version of BACI. Similar to concurrent programs, it is difficult to prove the correctness of distributed programs without an implementation. Distributed BACI allows distributed programs to be easily implemented. In addition to proving the correctness of a distributed program, one can use distributed BACI to test the program's performance. Distributed BACI is available at the following Web site: http://inside.mines.edu/fs_home/tcamp/baci/dbaci.html.
4. We have a PCODE disassembler that will provide the user with an annotated listing of a PCODE file, showing the mnemonics for each PCODE instruction and, if available, the corresponding program source that generated the instruction. This PCODE disassembler is included in the BACI System.
5. We have added the capability of separate compilation and external variables to both compilers (C and Pascal). The BACI System includes an archiver and a linker that enable the creation and use of libraries of BACI PCODE. For more details, see the BACI Separate Compilation User's Guide.

The BACI system has also been enhanced by others.

1. David Strite, an M.S. student who worked with Linda Null from the Pennsylvania State University, created a BACI Debugger: A GUI Debugger for the BACI System. This GUI is available at http://cs.hbg.psu.edu/~null/baci.
2. Using BACI and the BACI GUI from Pennsylvania State University, Moti Ben-Ari from the Weizmann Institute of Science in Israel created an

integrated development environment for learning concurrent programming by simulating concurrency called jBACI. jBACI is available at: http://stwww.weizmann.ac.il/g-cs/benari/jbaci.