Humboldt State University

Department of Computer Science

CS 374: Operating Systems

**Assignment 2 Sol**

Please upload solutions to the homework on Canvas. You can type answers directly into this document in the appropriate places.

1. Consider the Hoare monitor code below. Assume that 4 processes call the methods in the following order: hdl.Huey(), hdl.Louie(), hdl.Dewey(), hdl.Louie(). If multiple processes restart, execute Huey before Dewey before Louie. Please trace the execution of the code, and **note the values of the variables for each line of code executed** in the table given.

**monitor** hdl {

// The Huey, Dewey, and Louie monitor!

**int** money = 0, food = 0, mischief = 10;

condition trouble, mess, anger\_donald;

Huey( ) {

(1) money = money + mischief/5;

(2) if (mischief > 6) trouble.wait();

(3) mischief = mischief – 2; mess.wait();

(4) food++;

}

Dewey( ) {

(5) food = food + 2;

(6) if (mischief > 0) anger\_donald.signal();

(7) trouble.signal();

(8) mischief = mischief – 1;

}

Louie( ) {

(9) if (money > food) {

(10) food--; mischief++; anger\_donald.wait();}

(11) else mischief = mischief – 2;

(12) mess.signal();

(13) food = food + 3;

} }

(optional queues, to help tracing)

Condition Variable Queues:

**Trouble Mess Anger\_Donald**

~~Huey1~~ ~~Louie1~~

~~Huey1~~

High Speed Process Queues:

**Huey1 Louie1 Dewey1 Louie2**

~~Dewey1~~ ~~Dewey1~~

~~Louie2~~

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Line #:** | init | **1** | **2** | **9** | **10** | **5** | **6** | **12** | **13** | **7** | **3** | **9** | **11** | **12** | **4** | **8** | **13** |
| **food:** | 0 | **0** | **0** | **0** | **-1** | **1** | **1** | **1** | **4** | **4** | **4** | **4** | **4** | **4** | **5** | **5** | **8** |
| **money:** | 0 | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** | **2** |
| **mischief:** | 10 | **10** | **10** | **10** | **11** | **11** | **11** | **11** | **11** | **11** | **9** | **9** | **7** | **7** | **7** | **6** | **6** |

Some additional questions about hdl:

1. We did not go back to the other process once we were past the trouble.wait()in hdl.Huey(). Why not?

**Hoare semantics** say that we return to that other process upon the exit of hdl.Huey( ); we would only go back to the other process first under Java/Mesa semantics. We had a second wait call, the mess.wait(), which delayed the exit of hdl.Huey()

1. Why are none of the condition variables initialized to values like the integers are?

Condition variables are state variables with no value.

1. If the exit of a process might awaken multiple other processes, what determines their order of execution?

This depends on the order they were placed in the queue to await the exit of said process.

1. In this problem, all processes completed successfully. Is it possible that at the end of a series of calls, that some processes may not have completed? (HINT: where might they be if they didn’t complete?)

Yes, it is possible. Processes that are in wait queues might wait an indefinite period for some other process to execute an appropriate signal. They would be in wait states in our 5 state model.

1. Recall that each of the five philosophers, j, in the Dining Philosophers problem execute the following code segment:

P(j) { //possibly deadlocks!!

while (1) {

P(fork[j]);

P(fork[(j + 1) % 5];

eat;

V(fork[j]);

V(fork[(j + 1) % 5];

think( );

}

}

Suppose that instead of a particular fork being shared between a pair of philosophers, the five forks are placed in the middle of the table for all to share. Rewrite the code to take into account the forks are not distinct any more, and make sure you implement a simple protection mechanism to prevent deadlock (ignore starvation issues). You are free to use an if( P( ) ) test on a semaphore to do this.

Easiest way to do this: have each philosopher pick up two forks, but **only allow a maximum of two philosophers**. I put two versions that work below, and one that doesn’t. There are other ways, but they get trickier.

semaphore fork = 5; // this version **doesn’t** quite work

P(j){ // ***deadlocks possible*** again!!

while (1) {

P(fork); P(fork); eat; V(fork); V(fork); think( );

}

} // doesn’t work because each philosopher could grab 1 fork

semaphore fork = 5; diners = 2; **//** **this version works**

P(j){ // prevents deadlocks

while (1) {

P(diners); P(fork); P(fork); eat;

V(fork); V(fork); V(diners); think( );

}

}

semaphore fork = 5; **// this version uses if test on P() success**

P(j) { // assumes **if** test on P() available

while (1) {

P(fork);

if ( !P(fork) ) V(fork); // no second fork; start over

else { eat; V(fork); V(fork); } // got both forks

think();

}

}

1. The following expression describes the serial/parallel precedence relationship among several processes:

S( P( S( p1, p2 ), p3 ), p4, P( p5, S( p6, P( p7, p8 )) ) )

p7

p1

p2

p3

p5

p4

p8

p6

Transform this expression into a program using:

1. *Cobegin/coend* (or explain why this is not possible)

cobegin;

p1;

p2;

//

p3;

coend;

p4;

cobegin;

p5;

//

p6;

cobegin;

p7;

//

p8;

coend;

coend;

1. *Fork, join* and *quit* primitives

T1 = 2; T2 = 3;

Fork L1; Fork L3; quit;

L1: p1; fork L2; quit;

L2: p2; join t1, L4; quit;

L3: p3; join t1, L4; quit;

L4: p4; fork L5; fork p6; quit;

L5: p5; join t2, FIN; quit;

L6 p6; fork L7; fork L8; quit;

L7: p7; join t2, FIN; quit;

L8: p8; join t2, FIN; quit;

FIN: …

1. Below is a program expressed using *fork*, *join*, and *quit*.
   1. Draw the *process flow graph* for the program.

p1

p3

p2

p4

p5

p6

* 1. Either express the program using *S/P notation*, or explain clearly why this is not possible.

(10 points)

This cannot be expressed in S/P notation as the structure is not properly nested (see diagram under part a). Consider p4. Processes p1; p4 must run in parallel to p2. But p4; p6 must run parallel to p5. Hence this process p4 must be in two different parenthetical structures.

Another way to see this is via decomposing cuts. After the initial cut slicing off p3 as parallel to the rest of the processes, there are no further decomposing cuts. A vertical cut isn’t possible because of p4; a horizontal cut would slice through p2 or p5.

T1 = 2; T2 = 3;

fork L1; fork L2; fork L3; quit;

L1: p1; fork L5; fork L4; quit;

L2: p2; join T1, L6; quit;

L3: p3; join T2, FIN; quit;

L4: p4; join T1, L6; quit;

L5: p5; join T2, FIN; quit;

L6: p6; join T2, FIN; quit;

FIN: …

1. Consider the following process flow graph:

p6

p2

p4

p1

p3

p5

1. Express the graph as a single cobegin/coend block with all process coordination accomplished with semaphores.

Semaphore S1 = 0; S1B = 0; S2 = 0;

cobegin;

p1; V(S1); quit;

//

p2; V(S1); quit;

//

P(S1); P(S1); V(S1B); p3; V(S2); quit;

//

P(S1B); p4;

//

P(S2); p5;

//

p6;

coend;

Imagining real code that behaves like this diagram (assume ***matrix*** operations):

// C, D, and F are available at start

A = F\*C; // p1

B = C+D; // p2

Z = A/B + F; // p3

Y = A\*B – F; // p4

W = Z/C; // p5

M = F\*D/(C\*5) // p6

1. Express the graph using nested cobegin/coend blocks (no semaphores).

cobegin

p6;

//

cobegin

p1;

//

p2;

coend;

cobegin

p3; p5;

//

p4;

coend

coend

1. Consider the following rendezvous code:

while (1) {

select {

when a==TRUE :

accept A()

{ function1(); b = FALSE; }

when b==TRUE :

accept B()

{ function2(); a = FALSE; }

else { a = TRUE; b = TRUE; }

Assume that there aren’t outstanding calls to A or B when the code is first executed. Thereafter, the following calls arrive in the given order [in less time than it takes to do a pass through the code]: A( ), B( ), B( ), A( ), A( ), B( ).

1. In which order will the calls be accepted?

A, A, A, B, B, B.

1. Can a caller of A (or of B) be starved? If so, explain how.

Yes, starvation of either A or B can occur if an unbroken stream of the other is received.