Northern Michigan University (Marquette Co, MI)

CS444-01-25F: Parallel & Distributed Processing (Andrew A. Poe) Name: _____
Endterm Examination (Exam 2) Page 1/4 Wednesday 3 December 2025 9:00 A.M. EST

Answer all questions. Show all work.

Time: 50 minutes.

THE BEAR AND THE HONEYBEES: There is a honey pot initially empty. Bees arrive one at a time and release an amount of honey into the pot. When 1000 bees have deposited honey, the bear eats the honey. No two creatures can access the pot at the same time. No bee is allowed to deposit honey into a full pot. All creatures wait until they can use the pot. The bear runs an infinite loop, waiting until the pot is full, then eating the honey, and so forth. There can be any number of bees, they deposit their honey (after possibly waiting) and leave. Perhaps that same bee will return some day, perhaps not, but that's not your problem. There is no shortage of bees.

1. Emulate this problem with semaphores. Write two methods, one that emulates a bee and one that emulates the bear (the bear process contains an infinite loop). Use good passing the baton technique. A bee shouldn't wait if it doesn't have to, and no two creatures can access the pot at the same time. Use a bee semaphore, a bear semaphore, and an admin semaphore.

.

BEE

```
P (admin);
if (bear==1) {
 beew++;
 V (admin);
 P (BEE);
}
cout << "Dropping Honey" << endl;
honey++;
if (honey==1000) {
 bear = 1;
 V (BEAR);
} else if (beew > 0) {
 beew--;
 V (BEE);
} else V(admin)
```

BEAR

```
while (true) {
 P (BEAR);
 cout << "Eat honey" << endl;
 honey = 0;
 bear = 0;
 if (beew > 0) {
  beew--;
  V (BEE);
 else V(admin);
```

2. The same problem but use monitor code. Write a monitor to solve the problem and indicate how individual processes would use the monitor. Remember that no matter how many creatures there are, there is still only one monitor object.

```
BEE

if (bear==1) {
 beew++;
 wait (BEE);
}
cout << "Dropping Honey" << endl;
honey++;
if (honey==1000) {
 bear = 1;
 signal (BEAR);
} else if (beew > 0) {
 beew--;
 signal (BEE);
}

BEAR

while (true) {
 wait (BEAR);
 cout << "Eat honey" << endl;
 honey = 0;
 bear = 0;
 if (beew > 0) {
  beew--;
  signal (BEE);
}
```

3. The same problem but with message passing code. Have the master process emulate the bear and have all the other processes emulate bees. Write the code that the master process would run AND the code that the bee processes would run.

```
BEAR

while (true) {

 MPI_Recv (&msg,&proc);
 cout << "Honey has been delivered." << endl;y
 honey++;
 if (honey==1000) {
  honey = 0;
  cout << "Bear eats honey." << endl;
 }
}

BEE

MPI_Isend ("Honey",0);
```

4. Now, a BETTER version of the problem would allow as many bees as wanted to to deposit their honey simultaneously. (However, you still can't have bees and a bear at the same time.) With this alteration, what “classic” synchronization problem does this problem now resemble?

You don't have to write the code explicitly, but how would you have to modify your monitor code (problem 2) to emulate the modified problem? Remember that only one thread can be moving in the monitor at any given time. You don't have to write the code, but your answer still has to be specific, not vague.

This is very similar to the readers/writer problem. You can have as many readers as you want working simultaneously, but only one writer can write at a time, and it locks out all readers when it does.

Since only one thread can move in the monitor at any time, the code that can run simultaneously cannot be in the monitor.

Instead of a bee method in the monitor, you would need beebegin and beeend or something like that. beebegin would wait for the bear if necessary and then proceed while signalling another bee if possible (not if the honey pot is full).

Beeend would signal the bear if necessary or another bee if possible.

In between beebegin and beeend calls in your regular method you would have the code that would deposit honey. Since this code is not in the monitor, any number of bee processes can be running it.