

Monolithic Concurrency Control Consider a proposed concurrency control primitive, the *fubar*, with the following operations:

`fubar.sleep(boolean predicate)` – atomically tests predicate and, if false, blocks until the same object is signaled
`fubar.wakeup()` – wakes up –all- *fubars* sleeping on exactly the same object

Can *fubars* be used to ensure mutual exclusion (without deadlocking, livelocking, or other silliness) among two or more competitors for a single critical resource? If yes, please provide an example showing their use to protect some `criticalsection()` in competitors containing the following logic:

```
while (forever) {
    dosomestuff();
    criticalsection();
    dosomemorestuff();
}
```

If not, please explain the fundamental problem with this approach.

2. Consider a proposed concurrency control primitive, the *rabuf*, with the following operations:

`rabuf.sleep(mutex predicateMutex)` – always sleeps, placing caller into a FIFO queue for a later `signal()`, and atomically releases the mutex
`rabuf.signal()` – wakes up the head of the FIFO queue of those waiting for the *rabuf*

Can *rabufs* be used to ensure mutual exclusion exclusion (without deadlocking, livelocking, or other silliness) among two or more competitors for a single critical resource? If yes, please provide an example showing their use to protect some `criticalsection()` in competitors containing the following logic:

```
while (forever) {
    dosomestuff();
    criticalsection();
    dosomemorestuff();
}
```

If not, please explain the fundamental problem with this approach.

3. Consider a dinner table containing n plates, $2n$ forks, and unlimited servings of food (Ain't catering great?) . In order to eat, one needs to have 1 fork, 1 helping of food, and 1 plate. Consider a meal where each participant is starving and there is no nearby parental unit to enforce a communally productive sharing policy.

- (a) Assume that the mealtime logic is implemented —exactly— as below within each and every participant.

```
while (hungry) {
    breath();
    grab(fork, 1);
    grab(plate, 1);
    grab(food, 1);
    chowdown();
    burp();
}
```

Is deadlock possible? If so, in terms of n , how many participants are required before it becomes possible? How do you know?

- (b) Assume that the logic within each participant is implemented as below within each and every participant, but that each participant will reach for forks and plates in different orders, depending on which happens to be closer.

```
while (hungry) {
    breath();
    grab(fork, 1);
    grab(plate, 1);
    grab(food, 1);
    chowdown();
    burp();
}
```

Is deadlock possible? If so, please (a) identify, in terms of n , the minimum number of participants required before deadlock becomes a risk, and (b) illustrate using the code above as an example, how semaphores can be introduced to eliminate the risk of deadlock while ensuring progress.

4. Middleware/RPC/RMI

- (a) Consider the implementation of an RPC system in a homogenous environment (same hardware, same OS, same language, same, same, same). Is it possible to implement a pass-by-reference (not necessarily pass-by-address) mechanism? If not, why not? If so, in what ways might it be best to relax the semantics of a typical local pass-by-reference situation? Why?
- (b) Consider the implementation of an RPC system in a heterogeneous environment (different processor architecture, different OS, different programming language, different, different, different). How might the heterogeneity complicate the model? Please consider each of the following:
- i) Simple primitives (think back to 213 for how they can differ from system to system)
 - ii) Complex data types, including structs and strings,
 - iii) Higher-order language and library data structures, such as linked lists, maps, etc.
 - iv) Programming paradigms (function pointers, jump table, functors, etc)
- (c) Consider Java's RMI facility, which generates stubs at compile time. Could it, instead, generate the stubs at runtime? For example, could it disassemble a class file, or inspect an object's properties at runtime, rather than at compile time? If not, why not. If so, what would be the advantages and disadvantages of this model?

- (d) Consider Java's RMI facility, which only plays nicely with classes that implement the *Serializable* or *Remote* interfaces. Would it be possible to implement an RMI facility in Java that worked for all classes? For example, by using a combination of the class file, as well as reflection and other Java mechanisms to decompose, serialize, and reconstitute instances by brute force? If so, please explain any necessary limitations. If not, please example why not.

5. Distributed Concurrency Control

In class we discussed enforcing mutual exclusion, among other ways, via a central server, majority voting, and token ring.

- (a) Which of these systems requires the fewest messages under heavy contention? How many messages are required per request?

- (b) Which of these systems requires the most messages under heavy contention? Why?

- (c) Which of these systems is most robust to failure? Why?

