Slide 1

Administrivia

- (By e-mail — plans for upcoming reading quiz(zes), homework, exam.)

Slide 2

One More IPC Problem — Readers/Writers

- Scenario is that you have multiple processes wanting to access a shared file, some to read only but some to modify (write). Clearly(?) okay to allow multiple concurrent readers, but writers need exclusive access — i.e., there can be at most one writer at a time, and reading and writing at the same time is not allowed.

  (Right, a problem that immediately sounds useful!)

- Clearly need to employ one of the synchronization techniques discussed . . . Textbook shows one using semaphores, so look at that.

Slide 3

Readers / Writers — Solution

- We probably need at least two semaphores, one that will block readers when there's a writer active, and one that will block writers when there's either a writer or at least one reader.
- Seems like we also need some way to keep track of how many readers are active, and it's not clear how to do that using only semaphores. For the bounded-buffer problem we could use semaphores to give the right behavior without keeping an explicit count, but not clear how to do that here.
- So instead solution involves a shared integer count of readers (which we'll call `rc`) and two semaphores:
  - `mutex`, which will control access to `rc` and also(!) make readers wait if there's a writer active. Initial value is 1.
  - `db` (short for "database" — a common use case), to make everyone else wait if there's a writer active. Initial value is 1.

Slide 4

Readers / Writers — Code

- Shared variables as on previous slide.

Pseudocode for reader:

```
while (true) {
    down(mutex);
    rc += 1;
    if (rc == 1) down(db);
    up(mutex);
    /* read file */
    down(mutex):
    rc -= 1;
    if (rc == 0) up(db);
    up(mutex)
    /* use file data */
}
```

Pseudocode for writer:

```
while (true) {
    /* generate data */
    down(db);
    /* write file data */
    up(db);
}
```

Slide 5

Readers / Writers — Why It Works

- (Wasn't immediately obvious to me that it did — something about potentially blocking on `db` while holding something that looks like a mutex lock! But thinking carefully . . .)
- I like to think in terms of invariants — part of which is "what is this variable supposed to mean, and does it always mean that?"

  (Aside: I find this a *very* useful approach in general. To me it feels like a way of thinking about programs that may not occur to beginners but helps a lot.)

Slide 6

Readers / Writers — Why It Works, Continued

- `rc` can have any non-negative value; it represents a count of active readers.
- `mutex` can only have values 0 and 1. A value of 1 means there are no writers active and no readers actively trying to update `rc`. A value of 0 means either another reader is updating `rc` or another reader is waiting because there's a writer active.
- `db` can only have values 0 and 1. A value of 1 means there are no writers active, no readers active, andn no readers waiting because there's a writer active. A value of 0 means either there's one writer active or there's a reader active or there's a reader waiting because a writer is active.
- Proceeding informally, these do seem like invariants? Equally important, all possibilities meet the problem spec.

Slide 7

Readers / Writers — Potential Drawback

- A weakness of this approach is that it in effect gives readers high priority than writers — i.e., writers can wait forever.
- Fixable; the textbook gives a reference for a possibly-better solution, and it seems not hard to find them on the Web.

Slide 8

Minute Essay

- Questions?