Discussion questions for *A Micro-Kernel for Concurrency in C*

N. Ramsey and N. Bragg, for COMP 250RTS

October 4, 2017

Synchronization and communication

(1) How do `talk` and `listen` work? Walk through the code in Figure 1 on page 487 and explain the purpose and effect of each call to `p` or `v`.

(2) Review the implementation of semaphores on page 490. You may know that `p` is sometimes called "wait" and `v` is sometimes called "signal." I remind you that `p` decrements and `v` increments.

   (a) Informally, when a semaphore is represented as a small integer, what does the integer mean (or represent)?

   (b) Informally, when a semaphore is represented as a pointer to a linked list, what does the linked list represent?

   (c) What are the atomic operations on semaphores?

   (d) In Cormack's micro-kernel, how is atomicity enforced?

   (e) Why does Cormack choose this particular method of enforcing atomicity? What are the alternatives? What are the cost tradeoffs?

   (f) Quickly: In Python (if you know), how is atomicity enforced?

(3) Suppose you wanted to implement semaphores on a shared-memory multiprocessor with a sequentially consistent memory model.

   (a) What do you know about machine instructions used for synchronization or atomicity? Especially on the popular AMD64 and ARM platforms?

   (b) How would you use such instructions to implement a semaphore?

Bonus questions to take home

(4) On pages 487 and 488, Cormack sketches a model of distributed programming in which a Unix process acts like a virtual machine.[1] Within one virtual machine, each remote-communication channel (i.e., pipe) is served by a dedicated, local micro-process. But there is a problem here: if a micro-process executes a `read` or `write` system call, it is likely going to block the entire address space—and no other micro-process will run.

   How would you address this problem?

   Implementor's note: I have encountered a similar problem in a library running on top of pthreads, as recently as Spring 2017.

(5) After reading the descriptions of process state on pages 488 and 490, what do you think is the state of the currently running micro-process?

(6) Cormack's `emit` primitive specifies that a new micro-process is launched and it executes a new procedure call. Contrast this specification with the specification of Unix `fork`, which executes the child process in the *same* code as the parent process.

   (a) These design choices have implementation consequences. What are they?

   (b) Does each design choice make sense in its context (run-time system versus operating system)? Why or why not?

[1] Old style, as in IBM CP or VMware.