Eddie Gurnee
COSC 592
Operating Systems

Chapter 5 Reading Questions:

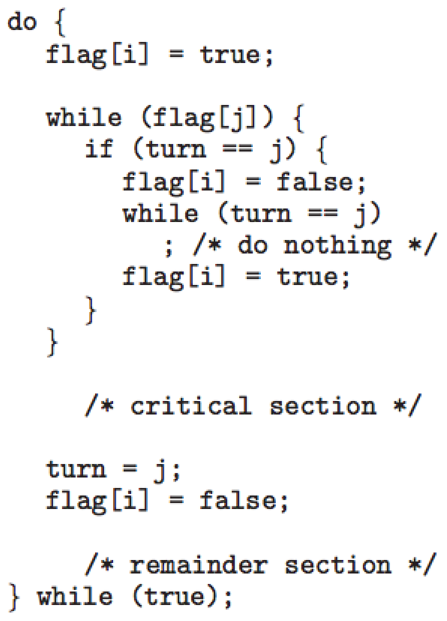
5.7) Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function and the wife calls `deposit()`. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

Response: The race condition would occur if the functions were invoked simultaneously and the account balance was to be adjusted separately by those two transactions; both of those processes need to ascertain what the current balance is in order to withdraw or deposit money into the account, if the balance is currently being edited by another transaction, the order of which the processing occurs will matter (race condition). To guard against this, it needs to be ensured that the account balance is only manipulated by one process at a time. This can be done by making each of the transactions synchronized.

5.8) The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, share the following variables:

```
    boolean flag[2]; /* initially false */
    int turn;
```

The structure of process P_i `(i == 0 or 1)` is shown in Figure 5.21. The other process is P_j `(j == 1 or 0)`. Prove that the algorithm satisfies all three requirements for the critical-section problem.



```
do {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; /* do nothing */
         flag[i] = true;
      }
   }

      /* critical section */

   turn = j;
   flag[i] = false;

      /* remainder section */
} while (true);
```

Figure 5.21 The structure of process P_i in Dekker's algorithm.

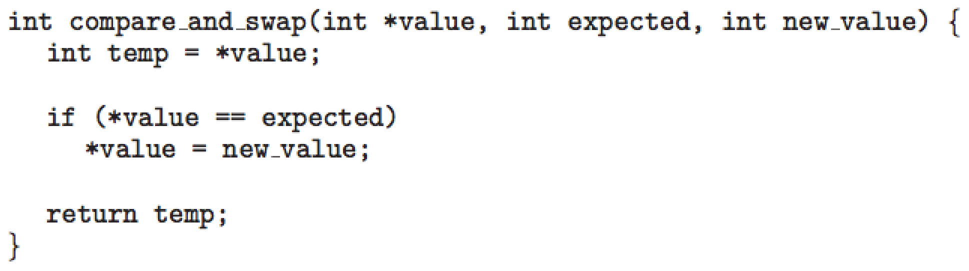
Response:

- Mutual Exclusion: referring the idea that only one process can be in its critical section at any given time, Dekker's algorithm forces the process j to wait until process i has finished its critical section therefore only one process can be in its critical section at any given time. (← by definition, mutually exclusive)
- Progress: only once the critical section of process i is complete does the turn variable switch to j;

with that being the case, only process that have not yet entered their remained section can decide who will enter the critical section next. (← by definition, progress is being done).

- Bounded Waiting: the process can only enter the critical section when the turn variable is set to the specific process; this waiting period is bounded by the fact that after the critical section of a specific process is finished, the turn variable is set to the other process, disallowing the first process to reenter its critical section until the second process changes the turn variable. (← by definition, bounded waiting).

5.14) Describe how the `compare_and_swap()` instruction can be used to provide mutual exclusion that satisfies the bounded–waiting requirement.



```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 5.5 The definition of the `compare_and_swap()` instruction.

Response: As the bounded-waiting requirement constrains the processes to a non-infinite wait time before entering the critical section, the processes that are attempting to access their respective critical sections should somehow have a way to represent the fact that they are waiting for access; this can be most simply achieved with a boolean array of the length of waiting processes. The code would look something like this (adapted from the examples in the book):

```
boolean lock; // initially set to false
boolean waiting[n]; // all initially set to 0
do {
      waiting[i] = true;
      int key = 1;
      while (waiting[i] && key) {
            key = compare_and_swap(&lock, false, true);
      }
      waiting[i] = false;

      /* critical section */

      j = (i + 1) % n;
      while ((j != i) && !waiting[j]) {
            j = (j + 1) % n;
      }
      if (j == i) {
            lock = false;
      } else {
            waiting[j] = false;
      }

      /* remainder section */

} while (true);
```

To be honest, I know there has to be a better, more elegant solution than this; I've just been wrestling with this for an hour and I need to move on to the other questions.

5.28) Discuss the tradeoff between fairness and throughput of operations in the readers–writers problem. Propose a method for solving the readers–writers problem without causing starvation.

Response: Throughput of operations can be maximized through use of multiple readers accessing the same data, though this focus on readers could result in starvation for the writers. This starvation could be avoided by implementing a priority queue for accesses: ordered by attempted access time, readers could jump in as long as a writer is not waiting, and upon completion of writing, access would be handed to the top item in the queue (the one that had been there the longest). Additionally, starvation could be avoided with proper dieting (and exercise) or through use of a Ring of Sustenance.

5.32) A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes currently accessing the file must be less than n. Write a monitor to coordinate access to the file.

Response:

```
monitor FileAccessMonitor {
      int currentSum = 0;
      const int nLIMIT;
      condition cond;

      void startAccess(int uniqueProcessNumber) {
            while (currentSum + uniqueProcessNumber >= nLIMIT) {
                  cond.wait();
            }
            currentSum += uniqueProcessNumber;
      }

      void endAccess(int uniqueProcessNumber) {
            currentSum -= uniqueProcessNumber;
            cond.broadcast();
      }
}
```