Pages 174-180, Problem numbers 1, 11, 32, 54, 55 and 56.

Name: Sarah Depew

Collaborators: TA session and Kellie Dinh

Resources Used: <https://www.quora.com/In-process-states-under-what-conditions-do-the-following-occur-running-to-ready-running-to-waiting-and-waiting-to-ready>, <https://docs.oracle.com/cd/E19455-01/806-5257/sync-34/index.html>

1. The two states that are not on this diagram are Ready 🡪 Blocked and Blocked 🡪 Running.

**Ready 🡪 Blocked** cannot occur, since this would clearly contradict what it means to be ready. Ready processes are not running and waiting for CPU time. A process that is not running cannot suddenly go into a blocked state, since the process would not be doing anything that would cause it to become blocked. This transition does not occur.

**Blocked 🡪 Running** would only be able to occur if the system bypassed the Ready state. In other words, this could only occur if the system had no waiting processes and so the process, as soon as it got what it was blocked for, could go onto the CPU. However, this situation is highly idealized and likely could not occur, since this would bypass the OS scheduler. Ergo, this transition also does not occur.

11) This solution is dependent upon the placement of the fork() statement. If the fork() is before the I/O input request, then the process can fork() and then two threads, due to interleaving, could end up requesting I/O at the same time. However, if the fork() happens after the I/O request, then this could not happen.

32) Let’s say we have a critical region of code that we want to protect, but we also want to let in MAX\_COUNT number of threads access at one time. (We still only want one thread to access the critical region at once.) A solution using only binary semaphores could look something like this:

typedef int semaphore;

int MAX\_COUNT = (some integer value);

semaphore mutex = 1;

semaphore controller = 1;

int count = MAX\_COUNT; //Number of resources available

// Some code here

down(&mutex) //exclusive access to count

if(count>0) count = count - 1; //one more resource taken away

else if(count == 0) down(&controller); //lock the area down if there are no resources available

up(&mutex); //release access to count

//Critical Region with MAX\_COUNT resources total

down(&mutex); //exclusive access to count

count = count + 1; //A resource has been freed

if(count <= MAX\_COUNT) up(&controller); //the section is available to multiple threads

up(&mutex); //release access to count

//More code here

This solution allows a maximum of MAX\_COUNT into the critical region, since there are only that many resources available.

54) The HUNGRY variable is eventually used in test() to know if the philosopher can get the forks needed to eat and can be put into an EATING state. Otherwise, if the philosopher is blocked from eating and not allowed to proceed. This HUNGRY variable denotes the philosopher’s intention to eat food and therefore a need for two forks.

55) In the function call to put\_forks(), neither the left nor the right neighbors would be permitted to eat until the *next* time they are asked if they want to eat. This is due to the fact that it would appear that philosopher is still using two forks when she (or he) is actually done using the forks. This would mean the solution is less fair, because for one round three people are not able to eat.

1. 56) **The readers and writers problem can be formulated in several ways with regard to which category of processes can be started when. Carefully describe three different variations of the problem, each one favoring (or not favoring) some category of proc- esses. For each variation, specify what happens when a reader or a writer becomes ready to access the database, and what happens when a process is finished.**

Situation 1: The situation we find in the book is one where readers are prioritized. In this solution, if there is a writer writing, the writer gets to write and then exit. However, if there is a reader reading from the database, any subsequent readers that come in to the database are allowed to join this first reader and read data. Clearly, this solution favors readers, since when a reader is ready to access the database they either can go immediately or, if there is a writer in the database, they must wait for the writer to finish and then go. This scenario paints a picture of unequal distribution of resources between the readers and the writers.

Situation 2: The situation where writers are prioritized.

Situation 3: The readers and writers are put onto a queue where they take turns writing and reading based upon the order their request came into the database.