Synchronization: Reader Writer Problem

Harrison Vuong (hvuong@ucsc.edu)
David Zou (dzou@ucsc.edu)
Derek Frank (dmfrank@ucsc.edu)

1 Goal:

The goal of reader writer is to use semaphores to eliminate starvation of the writers.

2 Available Resources:

Minix Documentations
Interprocess Communications and Synchronization Slides

3 Design:

void reader (int mutex, int writing, int priority, int readerid)

note: shared variables
number of reader file
time file for reader

open the time file for readers
generate a random number based on time
close the time file for readers

if the random number is below 20,000 + 20,000
pause for the given random number

semdown the priority
semdown the mutex

open then number of reader file
variable to keep number of readers
close the number of readers
increment the number of readers
open the number of reader file
write the new count of readers if any
close the number of reader file

if number of readers is 1
    semdown writing
    writer goes to sleep since there is a reader
semup the priority

print the reader is currently reading

generate a new random time
if new random time is less than 20,000 add 20,000 to it

open time file for readers
write the new time in file then close the file
decrease the number of readers
after opening and recording old number of readers
open and write new number of readers since reader has decreased

if there are currently no readers
    semup for the writers

semup the mutex
print that the reader is finished reading

int main (int argc, char **argv) → reader.c

check for correct usage
open the sem file
check for errors while opening
scan the contents of the semfile (which keeps track of currently used semaphores)
call reader with the retrieved semaphores

void writer (int mutex, int writing, int priority, int writerid)

create a writer time file to keep track of wait time
create a random number using random while using an algorithm to ensure that its between the given time for wait
usleep using the calculated random number
sem down on the priority
sem down on writing
    this is where the writing will take place
    show the writer is writing
sem up on writing
sem up on priority
    the writer is now done writing

int main (int argc, char **argv) → writer.c

check the usage of writer by checking options
set values for wid, mutex, writing and priority
open the semaphore file
do necessary error checking while opening
get values for writing and priority from the sem file
pass variables to writer for execution

int main (int argc, char **argv) → rwinit.c

create two file pointers
initialize mutex, priority and writing to 0
create the mutex semaphore
    do the necessary error checking using a switch to ensure that the semaphore is valid

create the writing semaphore
    do the necessary error checking using a switch to ensure that the semaphore is valid
create the priority semaphores
    do the necessary error checking using a switch to ensure validity again

open the sem file and write the semaphores to the file
close the file

open the number of readers and write 0
close the file

int main (int argc, char **argv) → rwfree.c

check to see if there is an error when opening sem file
    otherwise open the semfile
get the variables from the sem file
close the sem file
if semfree of the mutex is below 0, return error
if semfree of writing is 0 or below 0, return error
if semfree of priority is 0 or below 0, return error

delete number of readers file
delete reader time file
delete the semaphore file

4 Testing:

The main purpose of this test is to see if the usage of semaphores correctly. To prove this, writers are no supposed to starve, meaning, if there are many writers to readers, writers will eventually get priority to go over readers that entered after the writers. To do this, the test was to create multiple writer and reader processes running as background processes. In order to determine if the semaphores are working is to trace through which reader and which writer are getting access to which critical regions at a time.