Daniel Cender

Jan 26, 2020

CST-221 O500

Wk 3: Deadlock Avoidance

**Summary**

For this program, I envisioned a situation where a timer would be useless in avoiding thread deadlocks. I have 10 threads running in this program, with the first thread taking 10-seconds to complete. The other 9 only have a 4-second timeout before retrying. For a real-life scenario, maybe the longer-running thread is a chronologically triggered procedure on sensitive database items, whereas the short processes are a handful of small ETL (Extract, Transform, Load) jobs.

The program plays out pretty simply. The first thread gets the resource and processes it, while the other 9 threads start their timeout. The 9 threads’ timeouts finish, then they start another, since the first thread is still busy with its process. Once the longer thread ends, all the other threads are sleeping, so it can pick up the resource again and use it, locking it up for another 10 seconds.

I set a maximum of 5 runs for each thread process, so this pattern does not continue forever, as in the real world there would likely be some retry limit to minimize resource consumption or the user would quit the program.

**Design**

The Starvation program follows the flow outlined in this process diagram:

A close up of a map

Description automatically generated

**Result**

The output from this program is a plethora of console printouts that are written to a log.md file (markdown formatting). An unforeseen result of this function was that it successfully starved even the longer-running thread at times. Sometimes another thread would lock the mutex before the longer thread could, and it would starve out all other 9 successfully. Below are a couple screenshots of the final resulting output and the text from the resulting log.md file (past into <https://dillinger.io/> for proper Markdown formatting):

A screenshot of text

Description automatically generated

log.md output:

## Thread Starvation Program Log File

\_*Longer running thread now in control...*\_

\_*Thread 1 on its 1 iteration:*\_

- \_\_**Printing resource: 12345**\_\_

\_*Thread 3 putting itself to sleep on its 1 run*\_

\_*Thread 4 putting itself to sleep on its 1 run*\_

\_*Thread 6 putting itself to sleep on its 1 run*\_

\_*Thread 7 putting itself to sleep on its 1 run*\_

\_*Thread 8 putting itself to sleep on its 1 run*\_

\_*Thread 2 putting itself to sleep on its 1 run*\_

\_*Thread 9 putting itself to sleep on its 1 run*\_

\_*Thread 5 putting itself to sleep on its 1 run*\_

\_*Thread 10 putting itself to sleep on its 1 run*\_

\_*Thread 3 putting itself to sleep on its 2 run*\_

\_*Thread 4 putting itself to sleep on its 2 run*\_

\_*Thread 6 putting itself to sleep on its 2 run*\_

\_*Thread 7 putting itself to sleep on its 2 run*\_

\_*Thread 8 putting itself to sleep on its 2 run*\_

\_*Thread 2 putting itself to sleep on its 2 run*\_

\_*Thread 9 putting itself to sleep on its 2 run*\_

\_*Thread 5 putting itself to sleep on its 2 run*\_

\_*Thread 10 putting itself to sleep on its 2 run*\_

\_*Thread 3 putting itself to sleep on its 3 run*\_

\_*Thread 5 putting itself to sleep on its 3 run*\_

\_*Thread 4 putting itself to sleep on its 3 run*\_

\_*Thread 6 putting itself to sleep on its 3 run*\_

\_*Thread 8 putting itself to sleep on its 3 run*\_

\_*Thread 2 putting itself to sleep on its 3 run*\_

\_*Thread 9 putting itself to sleep on its 3 run*\_

\_*Thread 10 putting itself to sleep on its 3 run*\_

\_*Thread 7 putting itself to sleep on its 3 run*\_

\_*Longer running thread now in control...*\_

\_*Thread 1 on its 2 iteration:*\_

- \_\_**Printing resource: 12345**\_\_

\_*Thread 3 putting itself to sleep on its 4 run*\_

\_*Thread 5 putting itself to sleep on its 4 run*\_

\_*Thread 8 putting itself to sleep on its 4 run*\_

\_*Thread 4 putting itself to sleep on its 4 run*\_

\_*Thread 2 putting itself to sleep on its 4 run*\_

\_*Thread 9 putting itself to sleep on its 4 run*\_

\_*Thread 10 putting itself to sleep on its 4 run*\_

\_*Thread 7 putting itself to sleep on its 4 run*\_

\_*Thread 6 putting itself to sleep on its 4 run*\_

\_*Thread 3 putting itself to sleep on its 5 run*\_

\_*Thread 8 putting itself to sleep on its 5 run*\_

\_*Thread 4 putting itself to sleep on its 5 run*\_

\_*Thread 6 putting itself to sleep on its 5 run*\_

\_*Thread 5 putting itself to sleep on its 5 run*\_

\_*Thread 2 putting itself to sleep on its 5 run*\_

\_*Thread 10 putting itself to sleep on its 5 run*\_

\_*Thread 7 putting itself to sleep on its 5 run*\_

\_*Thread 9 putting itself to sleep on its 5 run*\_

\_*Longer running thread now in control...*\_

\_*Thread 1 on its 3 iteration:*\_

- \_\_**Printing resource: 12345**\_\_

---

### Thread 3 final results:

\_\_**Thread 3 got to run 5 times.**\_\_

\_\_**Thread 3 got to access RESOURCE 0 times**\_\_

---

---

### Thread 8 final results:

\_\_**Thread 8 got to run 5 times.**\_\_

\_\_**Thread 8 got to access RESOURCE 0 times**\_\_

---

---

### Thread 4 final results:

\_\_**Thread 4 got to run 5 times.**\_\_

\_\_**Thread 4 got to access RESOURCE 0 times**\_\_

---

---

### Thread 6 final results:

\_\_**Thread 6 got to run 5 times.**\_\_

\_\_**Thread 6 got to access RESOURCE 0 times**\_\_

---

---

### Thread 5 final results:

\_\_**Thread 5 got to run 5 times.**\_\_

\_\_**Thread 5 got to access RESOURCE 0 times**\_\_

---

---

### Thread 2 final results:

\_\_**Thread 2 got to run 5 times.**\_\_

\_\_**Thread 2 got to access RESOURCE 0 times**\_\_

---

---

### Thread 10 final results:

\_\_**Thread 10 got to run 5 times.**\_\_

\_\_**Thread 10 got to access RESOURCE 0 times**\_\_

---

---

### Thread 7 final results:

\_\_**Thread 7 got to run 5 times.**\_\_

\_\_**Thread 7 got to access RESOURCE 0 times**\_\_

---

---

### Thread 9 final results:

\_\_**Thread 9 got to run 5 times.**\_\_

\_\_**Thread 9 got to access RESOURCE 0 times**\_\_

---

\_*Longer running thread now in control...*\_

\_*Thread 1 on its 4 iteration:*\_

- \_\_**Printing resource: 12345**\_\_

\_*Longer running thread now in control...*\_

\_*Thread 1 on its 5 iteration:*\_

- \_\_**Printing resource: 12345**\_\_

---

### Thread 1 final results:

\_\_**Thread 1 got to run 5 times.**\_\_

\_\_**Thread 1 got to access RESOURCE 5 times**\_\_

---

**Reflection**

The use of timeouts to avoid deadlocks is a complex feature to implement correctly. The way I coded my solution illustrated some of the potential problems of using a timeout in a multi-threaded program. A small improvement would be to use a back off for the thread timeouts, so the time they sleep gets progressively longer until it hits a limit. An even better, though more resource-heavy, solution would be to replace the mutex check with a spinning lock. A busy wait in that fashion would catch the lock before the previously running thread locks it again.

A timeout like I implemented would be best suited for a situation like running testing suites, where a prior function will need to run with access to a resource, but when it gives it up, it won’t snatch it again and starve the waiting process/thread.

**Code**

Even though the entire code solution is below, it may also be found uploaded to my GitHub repository for this course, here: <https://github.com/DanielCender/CST-221/tree/master/Wk2/ProducerConsumer>

The below code was compiled from the command line and run with the following commands:

* ‘*gcc -o Starvation Starvation.c -lpthread*’, then
* ‘./*Starvation’*

/\*\*

*\**

*\* Author: Daniel cender*

*\* Date: 01/25/2020*

*\**

*\* Description: This program illustrates how using a timeout to restart threads*

*\* to avoid deadlock is a dangerous way to potentially have your program forever*

*\* awaiting conditions that will never be met.*

*\**

*\* In this program, we're setting up the scenario where the process at*

*\* threads[0] runs for longer than the other threads we're spinning up.*

*\**

*\* The short and long timeouts have been set to reflect a particular scenario.*

*\* With SHORT\_TIMEOUT == 4 and the LONG\_TIMEOUT == 10, we get a situation where*

*\* our shorter-running threads will run twice before the longer thread finishes it's first*

*\* run. It'll then start another 10second run before our shorter threads can lock the mutex.*

*\**

*\* So, in this case, our shorter threads never get access to their critical region of code, and their timeouts aren't intelligent enough to know they are futile.*

*\**

*\* Improvements: If I were to improve upon this so that all the threads could*

*\* gain access to the coveted resource, I would set a backoff for the sleep timeout.*

*\* If the timeout increased by 1 second every time, there'd be an eventual overlap*

*\* with the gap between the longer running thread's runs, so the starved thread could*

*\* hopefully lock the mutex in that time. Better yet, I'd implement a busy wait*

*\* instead of the current check, even though that requires more processing power.*

*\* A spinning lock would knab the resource at some point, but would likely still*

*\* distribute resource access unfairly.*

*\**

*\* Compiled and ran with:*

*\* ~ gcc -o Starvation Starvation.c -lpthread*

*\* ~ ./Starvation*

*\**

*\* Logs found in: /log.md*

*\**

\*/

#*include* <*limits.h*>

#*include* <*pthread.h*>

#*include* <*stdint.h*>

#*include* <*stdio.h*>

#*include* <*stdlib.h*>

#*include* <*string.h*>

#*include* <*unistd.h*>

/\* *The LONG\_TIMEOUT will be used in the first thread process,*

*then the first thread to encounter a currently locked mutex*

*will timout for the duration of SHORT\_TIMEOUT before trying again.*

*In this way, it'll test whether the critical resource is available,*

*but not with a busy-waiting lock.*

\*/

#*define* LONG\_TIMEOUT 10 // *this thread should hypothetically never go without the lock*

#*define* SHORT\_TIMEOUT 4 // *threads will "try" twice while longer thread runs*

#*define* NUM\_THREADS 10 // *set num of threads here*

#*define* THREAD\_ITERATIONS 5 // *set iterations to run in each thread created*

*pthread\_mutex\_t* mutex = PTHREAD\_MUTEX\_INITIALIZER;

*pthread\_t* threads[NUM\_THREADS] = { 0 };

*uint64\_t* resource = 12345; // *resource being "fought over"*

FILE*\** fptr;

*char\** cwd[PATH\_MAX];

/\*\*

*\* Basic implementation of Array.indexOf for int arrays*

\*/

*int64\_t* indexOf(*pthread\_t* *arr[]*, *pthread\_t* *el*)

{

*for* (*uint64\_t* i = 0; i < NUM\_THREADS; ++i) {

*if* (arr[i] == el) {

*return* i;

}

}

*return* *-*1;

}

*void\** thread\_func(*void\** *args*)

{

*uint64\_t* runsLeft = THREAD\_ITERATIONS;

*uint64\_t* nbrTimesAccessingResource = 0;

*pthread\_t* pid = pthread\_self();

*int64\_t* pidIndex = indexOf(threads, pid);

*while* (runsLeft > 0) {

*if* (pthread\_mutex\_trylock(*&*mutex) == 0) {

*if* (pid == threads[0]) {

printf("*Longer running thread now in control...\n\n*");

fprintf(fptr, "*\_Longer running thread now in control...\_\n\n*");

}

// *critical region*

printf("*Thread %lli on its %llu iteration:\n*", pidIndex *+* 1, (THREAD\_ITERATIONS *-* runsLeft) *+* 1);

fprintf(fptr, "*\_Thread %lli on its %llu iteration:\_\n\n*", pidIndex *+* 1, (THREAD\_ITERATIONS *-* runsLeft) *+* 1);

printf("*Printing resource: %llu\n*", resource);

fprintf(fptr, "*- \_\_Printing resource: %llu\_\_\n\n*", resource);

// *decrement remaining runs*

*--*runsLeft;

// *add to count of times this thread got to access the critical region*

++nbrTimesAccessingResource;

*if* (pid == threads[0]) {

// *First thread runs longer timeout while keeping mutex locked*

sleep(LONG\_TIMEOUT);

} *else* {

// *Might not need this actually*

sleep(SHORT\_TIMEOUT);

}

// *give up mutex lock*

*if* (pthread\_mutex\_unlock(*&*mutex) != 0) {

perror("*Couldn't unlock mutex!\n*");

pthread\_exit(*&*pid);

}

} *else* {

printf("*Thread %lli putting itself to sleep on its %i run\n*", pidIndex *+* 1, (THREAD\_ITERATIONS *-* runsLeft) *+* 1);

fprintf(fptr, "*\_Thread %lli putting itself to sleep on its %i run\_\n\n*", pidIndex *+* 1, (THREAD\_ITERATIONS *-* runsLeft) *+* 1);

// *Also, decrement here: some threads need to starve so others can eat*

*--*runsLeft;

// *set timeout for shorter processes*

sleep(SHORT\_TIMEOUT);

}

}

// *log all final values*

printf("*\n---\n### Thread %lli final results:\n\n\_\_Thread %lli got to run %llu times.\_\_\n\n\_\_Thread %lli got to access RESOURCE %llu times\_\_\n\n---\n*", pidIndex *+* 1, pidIndex *+* 1, THREAD\_ITERATIONS *-* runsLeft, pidIndex *+* 1, nbrTimesAccessingResource);

fprintf(fptr, "*\n---\n### Thread %lli final results:\n\n\_\_Thread %lli got to run %llu times.\_\_\n\n\_\_Thread %lli got to access RESOURCE %llu times\_\_\n\n---\n*", pidIndex *+* 1, pidIndex *+* 1, THREAD\_ITERATIONS *-* runsLeft, pidIndex *+* 1, nbrTimesAccessingResource);

*return* *NULL*;

}

*int* main()

{

// *Get current working directory*

// *Ref: https://stackoverflow.com/questions/298510/how-to-get-the-current-directory-in-a-c-program*

*if* (getcwd(cwd, *sizeof*(cwd)) == *NULL*) {

perror("*getcwd() error*");

*return* 0;

}

// *Open/create new log file - will re-write file on every run*

*if* ((fptr = fopen(strcat(cwd, "*/log.md*"), "*w*")) == *NULL*) {

perror("*File could not be opened or created!*");

*return* 0;

}

fprintf(fptr, "*## Thread Starvation Program Log File\n\n*");

// *Create threads*

*for* (*uint64\_t* idx = 0; idx < NUM\_THREADS; ++idx) {

pthread\_create(*&*(threads[idx]), *NULL*, thread\_func, *NULL*);

}

// *Join threads to main*

*for* (*uint64\_t* idx = 0; idx < NUM\_THREADS; ++idx) {

pthread\_join(threads[idx], *NULL*);

}

*return* 0;

}

References

C File Handling. (n.d.). Retrieved January 25, 2020, from <https://www.programiz.com/c-programming/c-file-input-output#opening>

How to C in 2016. (n.d.). Retrieved January 23, 2020, from https://matt.sh/howto-c

Tanenbaum, A. S., & Bos, H. (2017). *Modern operating systems*. Vancouver, B.C.: Langara College.