The Widget Factory

This section problem comes from an old handout written by Julie

Fixing the factory (ungraded)

The following program simulates the operations of a widget factory. There are 26 "Worker" threads (one for each letter of the alphabet) who each have the task of making a batch of 100 widgets and one "Supervisor" thread who watches over the factory operations and encourages the workers along until all the widgets are made. The first three finishers get special recognition at the end of the simulation. The full program is reproduced later in this handout.

First, just look over the program and make some observations. There are several global variables accessed by multiple threads and absolutely no semaphores, so that by itself should raise some suspicion. After section, try running the program several times and note who the "winners" are. You should get different results on each run, and often the program can hang or crash because of its buggy state.

This kind of non-determinacy is the beauty and the frustration of concurrent program. Even when correctly synchronized, the results can vary widely because of random scheduling patterns.

Now let's dissect the program and its problems and put you to work fixing them. For each issue enumerated below, you'll investigate the problem, analyze the root cause, and go on to correct it.

The winners race condition in Worker

> You can most reliably get the buggy program to run to completion without hanging midway or crashing by running on an epic or elaine. Run the program repeatedly and you'll find that sometimes one or more entries in the `winners` array prints as "Empty". Estimate how frequently this problem occurs by counting how often it happens in the first 10 or so trials. Trace the code over carefully and figure out how the error occurs by devising a scheduling pattern between threads that will cause it. Try to force the bug to deterministically show up by adding in a `ThreadSleep` call that moves the thread off the processor at the most inopportune time. At this point, you should have a good idea of where the critical section is. Identify what needs protecting and add a semaphore into the program to control access to the critical region. This is a canonical example of using a binary semaphore as a lock to serialize access to global data.

On rare occasions this race condition can have an even nastier effect that crashes the program with a segmentation fault. Can you identify the extreme case that causes this to happen?

The numWidgetsMade race condition in Worker

The Worker threads also conflict in their access of the global `numWidgetsMade`. It is true that exactly 26*100 widgets are made (i.e. there are 2600 calls to the `MakeWidget` function) no matter what— explain why that is. But is it guaranteed that the global counter will be 2600 when done? Add a statement to print out the value of `numWidgetsMade` at the end. If you run on the elaines or epics, the program will usually complete and 2600 will be printed. Now try running the program on a saga. It most likely won't finish at all, but will hang midway. Run the program under the debugger and when it gets stuck, try to sort out what is happening. Which threads haven't completed? What is the value of numWidgetsMade? Run the program 10 or so times on a saga and note how often numWidgetsMade doesn't have the value 2600. Now think through the issue and construct a scheduling pattern that causes the problem. This leads us to another critical region that needs protecting. We could lock the region using the same semaphore we created above, but why it would be better to create a separate lock? Edit the program and add a new semaphore to correctly protect the critical code. Does it now end at 2600 reliably on all machines?

Whereas the epics and elaines are uniprocessors, the saga machines have two CPUs. On a single processor, the likelihood that the threads are swapped out in the tiny, tiny section that is critical is so unlikely that it effectively never happens. Yet with two processors, the likelihood the two of them are working in tandem in the critical region is almost unavoidable. This brings up a good point: test your program on both, because some bugs may only show up on one or the other.

On rare occasions, the numWidgets counter can end up as an unreasonably large value as a side-effect of the race condition described above in the winners array. Can you map out the connection between cause and effect? Where did that large number come from?

Busy-waiting in the Supervisor

The Supervisor loops, printing an encouraging message each time the total widgets made reaches another multiple of 100. Look carefully at the inner loop. It cycles until the current total is *at least* the target multiple. What if we wanted to announce *exactly* as we reached the multiple? Change the test in the while loop to be while `numWidgetsMade != target` and re-run the program. What happens? Explain the result that you see.

This inner loop is a classic spin-lock. You should regard all such "busy-waiting" as evil, it squanders processing time in senseless polling. You might think it would be less egregious if you added a sleep delay in the loop body. This would cause it to check less frequently, but that's just "slightly-less-busy waiting" and still not desirable. To get the Supervisor to wake up at the target multiple without busy-waiting, have it wait on a semaphore that only gets signaled when `numWidgetsMade` gets to a multiple of 100. This is a chance to use a semaphore as a rendezvous. When using a binary lock, one thread does both the wait and signal, but in a rendezvous, one thread waits on a semaphore that is signaled by a different thread as a means of communication.

Create a rendezvous semaphore between the Workers and Supervisor to eliminate the busy-waiting. Does this semaphore guarantee the Supervisor will print the encouraging message exactly at the target multiple, no more, no less? What would it take to guarantee that?

A race condition in the Supervisor

After all the widgets have been made, the Supervisor prints out the top three finishers. The Supervisor is supposed to wait until all Workers have exited before printing the list, but it currently prints the results right after it sees that 2600th widget is made. It is possible for the list to be printed before all the Worker threads have printed their own "Done" message, and in fact, it's even possible for the Supervisor to try to print out the winners' names before they have been assigned in the array. Describe a scheduling pattern that would cause this to happen. Run 10 trials and report how frequently one or more Workers prints “Done” after the list has been printed. (The error occurs almost never on a saga, and rarely on epic or elaine, so you may have trouble seeing it.)

Add in one more semaphore, this one to be used as a generalized rendezvous semaphore. Rather than alternating between the values 0 and 1, a general semaphore can take on any non-negative value. Use this semaphore to "count" the number of worker threads who have finished. Have the Supervisor watch as each Worker thread finishes and only print out the winners when all Workers are done. Even if one or more of the Worker threads signals this counter before the Supervisor starts watching it, it still works out correctly— why is that?

```
/*
 * factory.c
 * ---------
 * This concurrent program is designed to simulate a simple widget factory.
 * There are 26 "Worker" threads (one for each letter of the alphabet)
 * and each worker is builds a batch of 100 widgets, thus a total of
 * 2600 widgets will be built overall. Each time a Worker makes a widget,
 * it updates the global count of the number of total widgets built. The
 * one "Supervisor" thread keeps an eye on this global counter and each time
 * it reaches a multiple of 100, the Supervisor prints an encouraging
 * messages to the workers to keep their productivity up. There is also
 * an array which keeps track of the names of the first 3 worker threads
 * that finish, and the supervisor announces those names after all the
 * widgets have been built at the end of the simulation.
 *
 * As it stands, the code has some definite problems.  It uses no semaphores
 * and makes no attempt to avoid contention for any globals accessed by
 * multiple threads.  Following the instructions on the handout, you will
 * observe the behavior of the buggy program and insert the necessary
 * semaphores to fix it up until it runs perfectly every time!
 */

#include "thread_107.h"
#include <stdio.h>
#include <stdlib.h>

#define NUM_WORKERS 26  // one for every letter in the alphabet
#define NUM_WINNERS 3
#define BATCH 100

static void RandomDelay(int atLeastMicrosecs, int atMostMicrosecs);
static void MakeWidget(void);
static void Worker(void);
static void Supervisor(void);

static const char *winners[NUM_WINNERS] = {"Empty", "Empty", "Empty"};
static int numWorkersDone = 0, numWidgetsMade = 0;

/*
 * The main creates all the worker threads and the one supervisor
 * thread and then lets them run to completion.  They should all finish
 * when the simulation is done. By running with the -v flag, it will
 * include the trace output from the thread library.
 */


int main(int argc, char **argv)
{
    int i;
    char name[32];
    bool verbose = (argc == 2 && (strcmp(argv[1], "-v") == 0));

    InitThreadPackage(verbose);

    for (i = 0; i < NUM_WORKERS; i++) {
        sprintf(name, "Worker %c", i + 'A');
        ThreadNew(name, Worker, 0);
    }
    ThreadNew(“Supervisor”, Supervisor, 0);
```

```
    RunAllThreads();            // Let them all loose
    return 0;
}

/*
 * Supervisor
 * ----------
 * This function is executed by one single Supervisor thread. The
 * Supervisor mostly just hangs out and watches the Workers toil away.
 * However, in order to keep everyone's productivity up, the supervisor
 * will note when each multiple of 100 widgets has been reached and
 * prints an encouraging message. Since each worker makes 100, this
 * should happen NUM_WORKERS times.  After all the widgets have been
 * made, the supervisor reports who the top 3 finishers were and maybe
 * they get a nice raise. :-)
 */

static void Supervisor(void)
{
    int i, target;

    for (target = BATCH; target <= NUM_WORKERS*BATCH; target += BATCH) {

         while (numWidgetsMade < target) ;   // wait for multiple of 100
         printf("%s says: Good work, folks, we've made %d!\n",
                   ThreadName(), target);
    }

    printf("\n%s says: Super! We're done. The top finishers were:\n",
            ThreadName());

    for (i = 0; i < NUM_WINNERS; i++)
         printf("  %d) %s\n", i + 1, winners[i]);

}

/*
 * Worker
 * -------
 * This function is executed by each of the 26 Worker threads.  Each
 * worker is charged with the task of creating 100 widgets (using the
 * MakeWidget function) and updating the global counter after each one.
 * When finishing making all 100 widgets, the Worker also increments the
 * count of finished workers, and if one of the first few finishers, it
 * records its name in the list of winners.
 */

static void Worker(void)
{
    int i;

    for (i = 0; i < BATCH; i++) {
         MakeWidget();
         numWidgetsMade++;
    }

         /* When done, we want to increment counter of num finished workers,
          * but only store name in list if we were one of the top finishers */
    if (numWorkersDone++ < NUM_WINNERS)
```

```
        winners[numWorkersDone-1] = ThreadName();
    printf("%s DONE!\n", ThreadName());
}

/*
 * MakeWidget
 * ----------
 * Just inserts a random delay to vary execution patterns as a means
 * of simulating that the thread goes off and does some independent
 * activity here.
 */

static void MakeWidget(void)
{
    RandomDelay(5, 10);   // sleep random amount to simulate making pat
}

/*
 * RandomDelay
 * -------------
 * This is used to put the current thread to sleep for a little bit.
 * This function is already thread-safe and should require no modification.
 */

static void RandomDelay(int atLeastMicrosecs, int atMostMicrosecs)
{
    extern long random();
    long choice;
    int range = atMostMicrosecs - atLeastMicrosecs + 1;

    PROTECT(choice = random());      // protect non-re-entrancy
    ThreadSleep(atLeastMicrosecs + choice % range); // put thread to sleep
}
```

Fixing the factory—Identifying the Problems and Discussing the Fixes

Worker winner array race condition: About a third to half the time "Empty" shows up as a winner. (Your mileage may vary-- experimental results are dependent on the machine type, current load, and whim of the scheduler) This happens when a thread checks the value of `numWorkersDone` but is swapped out before it writes its name in the array. By the time it gets back on the processor, `numWorkersDone` was incremented by others and thus it writes its name off the end of the array somewhere and leaving Empty in its slot. A simple binary semaphore used to lock the critical region will ensure that a worker thread checks, increments, and assigns into the array without interference from other worker threads.

The truly extreme result is that when it writes too far off the end, it overwrites the neighboring variables `numWorkersDone` and `numWidgetsMade` which can lead to confusion and serious crashes later.

Worker numWidgetsMade racec condition: Each worker thread loops exactly 100 times. It uses a local stack variable as the counter and the upper bound on the loop comes from a compile-time constant so there is no concern about interference from other threads affecting the loop. Since there are 26 worker threads, we get exactly 2600 calls to `MakeWidget`.

Each worker thread increments the global counter each time through the loop, so you might also conclude that those 2600 increment operations guarantee the result has to be 2600, but the C statement `numWidgetsMade++` is not atomic. From our code generation work, we know that it takes about three instructions: load the value from memory into a register, do an add operation, and then store it back. What happens when a worker thread loads the value, gets swapped out, and meanwhile other threads come along and increment the counter? When the previous thread gets back on the processor, it will store a stale value for the counter. The range of possible results can be as low as 100, but never more than 2600. This race condition is much more rare, in fact, on a uniprocessor machine (such as the elaines or epics) you effectively never see this error because the window of opportunity is so tiny. However, on the multi-processor machines (saga) it shows up almost every time, because two threads running simultaneously on the two CPUs can be loading and storing at same time. In those cases, the result ends up being close, but not quite 2600, causing the Supervisor to loop endlessly.

All that's needed is another binary semaphore to lock the critical statement. Reusing the same lock from above would unnecessarily reduce concurrency since the two critical regions are not related, so a separate semaphore is warranted.

The unexpected large value for numWidgetsMade comes from the problem

described above with overrunning the winners array and trashing the neighboring variables.

Supervisor busy-waiting: After changing the test to equality, it is almost sure that the supervisor thread will infinitely loop since it is not likely to get processor time exactly at the moment the value is changed to a multiple of 100 but before it is incremented past that.

The rendezvous semaphore to fix this should be initialized to 0 (since it starts as not yet available) and the supervisor will wait on it. Since its value is 0, the supervisor immediately blocks. When a worker thread increments the global count, it checks for a multiple of 100, and if so, signals the supervisor to wake up and print the encouraging message. (Note that it is important that the worker threads do this under protection of the earlier lock we put in place to avoid double-signaling the supervisor). This rendezvous semaphore doesn't guarantee that the supervisor will immediately wake up and print at that time, it still depends on variation of the scheduler.

Supervisor race condition: Since all 2600 widgets have to have been made before the supervisor prints the winners, you know that all the worker threads have to be out or on their way out of the loop, but you can't be sure they are totally done, since they could be swapped out after the loop but before finishing. About one in 20 times, a worker will have not yet printed DONE before the Supervisor prints the results.

A generalized rendezvous semaphore can be used to count the number of workers finished. It is initialized to 0 and signaled once by each exiting worker. The supervisor waits on this semaphore in a loop 26 times, "counting" the number of finished workers until all have checked in. It's no problem if a worker thread signals before the supervisor gets around to waiting on it, because the semaphore retains its count and knows how many times it has been signaled. The supervisor will quickly note all previously finished threads and then efficiently wait for any remaining workers to signal.