Concurrent Programming

Annalise Tarhan

October 16, 2020

12 Homework Problems

12.16 Write a version of hello.c that creates and reaps n joinable peer threads, where n is a command-line argument.

```
int main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: %s \n", argv[0]);
                exit(0);
        }

        int num_threads = atoi(argv[1]);
        pthread_t tids[num_threads];

        for (int i = 0; i < num_threads; i++) {
                pthread_t tid;
                pthread_create(&tid, NULL, thread, NULL);
                tids[i] = tid;
        }

        for (int i = 0; i < num_threads; i++) {
                pthread_join(tids[i], NULL);
        }

        exit(0);
}
```

12.17

12.17.1 The program below has a bug. The thread is supposed to sleep for 1 second and then print a string. However, when we run it on our system, nothing prints. Why?

When one thread in a process exits, all the other threads in that process are killed. The sleep call guarantees that the first thread will exit before the string is printed.

12.17.2 You can fix this bug by replacing the exit function with one of two different pthreads function calls. Which ones?

pthread_join and pthread_detach

12.18 Using the progress graph, classify the following trajectories as either safe or unsafe.

12.18.1 H_2, L_2, U_2, H_1, L_1, S_2, U_1, S_1, T_1, T_2

unsafe

12.18.2 H_2, H_1, L_1, U_1, S_1, L_2, T_1, U_2, S_2, T_2

safe

12.18.3 H_1, L_1, H_2, L_2, U_2, S_2, U_1, S_1, T_1, T_2

unsafe

12.19 Derive a solution to the first readers-writers problem that gives stronger priority to readers, where a writer leaving its critical section will always restart a waiting reader if one exists.

Use an additional semaphore for writers. In addition to the mutex semaphore that protects readcnt and the w semaphore that writers the first reader in lock, add an extra W semaphore that only writers lock and unlock. Writers must lock W before they can lock w, and they unlock w before they unlock W. Since readers don't have to wait for W to be unlocked, they can lock w before another writer is able to.

```
sem_t mutex, w, W;

void writer(void)
{
        while (1) {
                P(&W);
                P(&w);
                 ...
                V(&w);
                V(&W);
        }
}
```

12.20 Derive a solution to the readers-writers problem where there are at most N readers that gives equal priority to readers and writers, in the sense that pending readers and writers have an equal chance of being granted access to the resource.

Credit due elsewhere. With the limit on the number of readers, access becomes a scarce resource, even for readers. They take one slot at a time, but this solution allows a waiting writer to claim the slots as well. Once the writer has acquired all N slots, it is free to write, since all of the readers are locked out. When it is finished, it releases all N slots. Writers don't compete, since there is a mutex guarding the slot-claiming routine.

```
int N;
sem_t mutex; /* Initially = 1 */
sem_t slots; /* Initially = N */

void reader(void)
{
        P(&slots);
        /* Critical section */
        V(&slots);
}

void writer(void)
{
        P(&mutex);
        for (int i = 0; i < N; i++) {
                P(&slots);
        }
        /* Critical section */
        for (int i = 0; i < N; i++) {
                V(&slots);
        }
        V(&mutex);
}
```

12.21 Derive a solution to the second readers-writers problem, which favors writers over readers.

This solution is inefficient, since it doesn't allow more than one reader at a time. It sets up two semaphores for readers and only one for writers. Only one of any kind is allowed access to the resource at a time, and since the readers have to pass an extra hurdle, it favors writers.

```
void reader(void)
{
        P(&reader_gate);
        P(&mutex);
        /* Critical section */
        V(&mutex);
        V(&reader_gate);
}

void writer(void)
{
        P(&mutex);
        /* Critical section */
        V(&mutex);
}
```

12.22 Modify the server so that it echoes at most one text line per iteration of the main server loop.

Instead of connecting and reading from connfd directly, this version connects and adds connfd to the read set. On the subsequent iteration, it will start reading. It also calls echo_line instead of echo, which returns 1 if it reached EOF and 0 otherwise. When echo_line returns 1, it removes connfd from the read set and closes the connection. Otherwise, it carries on to the next iteration and reads another line.

```
int has_open_connection = 0;

while (1) {
        ready_set = read_set;
        select(listenfd+2, &ready_set, NULL, NULL, NULL);
        if (FD_ISSET(STDIN_FILENO, &ready_set))
        command();
        if (has_open_connection) {
                if (echo_line(connfd)) {
                        close(connfd);
                        FD_CLEAR(connfd, &read_set);
                        has_open_connection = 0;
                }
        }
        if (!has_open_connection
        && FD_ISSET(listenfd, &ready_set)) {
                clientlen =
                        sizeof(struct sockaddr_storage);
                connfd = accept(listenfd,
                        (SA *)&clientaddr, &clientlen);
                FD_SET(connfd, &read_set);
                has_open_connection = 1;
        }
}
```

12.23 The event-driven echo server is flawed because a malicious client can deny service to other clients by sending a partial text line. Write an improved version of the server that can handle these partial text lines without blocking.

I suppose it would be cheating to throw each readline in its own thread?

12.24 The functions in the RIO I/O package are thread-safe. Are they reentrant as well?

They are reentrant, but only implicitly since some of the arguments are pointers to what could be shared data.

12.25 In the prethreaded concurrent echo server, each thread calls the echo_cnt function. Is echo_cnt thread-safe? Is it reentrant? Why or why not?

The function is thread-safe but not reentrant. It isn't reentrant because it references static global variables. It uses a semaphore to protect shared variables, though, so it is thread-safe.

12.26 Use the lock-and-copy technique to implement a thread-safe non-reentrant version of gethostbyname called gethostbyname_ts.

```c
struct hostent *gethostbyname_ts(const char *name)
{
        struct hostent *sharedp;
        struct hostent *copied_hostent = NULL;

        P(&mutex);
        sharedp = gethostbyname(name);
        copied_hostent->h_name = sharedp->h_name;
        copied_hostent->h_aliases = sharedp->h_aliases;
        copied_hostent->h_addrtype = sharedp->h_addrtype;
        copied_hostent->h_length = sharedp->h_length;
        copied_hostent->h_addr_list = sharedp->h_addr_list
        V(&mutex);
        return copied_hostent;
}
```

12.27 Why does the following approach for reading and writing sockets, which opens two standard I/O streams on the same open connected socket descriptor, create a deadly race condition in a concurrent server based on threads?

Each fclose operation attempts to close the same underlying socket descriptor. In a sequential program, this isn't a problem, but in a concurrent one that socket descriptor could have already been assigned to a different, newly opened socket and the second fclose operation could close that socket instead.

12.28 Does swapping the order of the two V operations have any effect on whether or not the program deadlocks?

No it doesn't. It restricts the possible trajectories through the state space, but none of the orderings lead to deadlock. Deadlock would only occur if the threads' P operations were in different orders.

12.29 Can the following program deadlock? Why or why not?

No it can't. By the mutex lock ordering rule, the program is deadlock-free since each thread acquires mutexes in order and releases them in reverse order.

12.30 Consider the following program that deadlocks.

12.30.1 For each thread, list the pairs of mutexes that it holds simultaneously.

Thread 1: (a,b) (a, c)

Thread 2: (c, b)

Thread 3: (b, a)

12.30.2 If $a < b < c$, which threads violate the mutex lock ordering rule?

Threads 2 and 3. Thread 2 acquires c before b and thread 3 acquires b before a. Technically, thread 1 releases b before acquiring c and threads 2 and 3 acquire c before anything else, but those infractions are less significant.

12.30.3 For these threads, show a new lock ordering that guarantees freedom from deadlock.

Given the order $a < b < c$, the perfect lock ordering is P(a); P(B); P(C); V(C); V(B); V(A).

12.31 Implement a version of the standard I/O fgets function, called tfgets, that times out and returns NULL if it does not receive an input line on standard input within 5 seconds. It should use processes, signals, and nonlocal jumps.

```
jmp_buf buf;

void sigchld_handler(int sig)
{       longjmp(buf, 1);        }

char *tfgets(char *s, int size, FILE *stream) {
        if (fork() == 0) {
                sleep(5);
                exit(0);
        }
        signal(SIGCHLD, sigchld_handler);
        if (setjmp(buf)) {
                return NULL;
        } else {
                return fgets(s, size, stream);
        }
}
```

12.32 Implement another version of the tfgets function that uses the select function.

```
char *tfgets(char *s, int size, FILE *stream)
{
        struct timeval tv;
        tv.tv_sec = 5;
        tv.tv_usec = 0;

        fd_set read_set;
        FD_ZERO(&read_set);
        FD_SET(STDIN_FILENO, &read_set);

        if (select(1, &read_set, NULL, NULL, &tv)) {
                return fgets(s, size, stream);
        } else {
                return NULL;
        }
}
```

12.33 Implement a threaded version of the tfgets function.

```
int complete = 0;
char *result = NULL;

struct fgets_args {
        char *s;
        int size;
        FILE *stream;
};

void *thread1(void *vargp) {
        sleep(5);
        complete = 1;
        return NULL;
}

void *thread2(void *vargp) {
        struct fgets_args *args =
                (struct fgets_args *) vargp;
        result = fgets(args->s, args->size, args->stream);
        complete = 1;
        return NULL;
}

char *tfgets(char *s, int size, FILE *stream)
{
        struct fgets_args args;
        args.s = s;
        args.size = size;
        args.stream = stream;

        pthread_t tid1, tid2;
        pthread_create(&tid1, NULL, thread1, NULL);
        pthread_create(&tid2, NULL, thread2, &args);
        while (!complete) {}
        return result;
}
```

12.34 Write a parallel threaded version of an N x M matrix multiplication kernel. Compare the performance to the sequential case.

This version creates a new thread for each entry in the final N x N matrix, which is extremely inefficient. For a 10 x 10 matrix, the concurrent version ran almost a thousand times slower. A (much) better approach would have been to divide the matrix into around four parts and split the work between as many threads. Actually, this is probably a great example of code I'll be absolutely horrified to look at in a few years.

```c
struct coordinates {
        int i;  int j;
};
void thread_mul()
{
        int i, j;
        pthread_t tids[10][10];
        struct coordinates crds[10][10];
        for (i = 0; i < 10; i++) {
                for (j = 0; j < 10; j++) {
                        struct coordinates coords;
                        coords.i = i;
                        coords.j = j;
                        crds[i][j] = coords;
                        pthread_create(
                                &tids[i][j], NULL,
                                thread, &crds[i][j]);
                }
        }
        for (i = 0; i < 10; i++) {
                for (j = 0; j < 10; j++) {
                        pthread_join(tids[i][j], NULL);
                }
        }
}
void *thread(void *vargp) {
        int i = ((struct coordinates *) vargp) -> i;
        int j = ((struct coordinates *) vargp) -> j;
        int sum = 0;
        for (int k = 0; k < 8; k++) {
                sum += A[i][k]*B[k][j];
        }
        C[i][j] = sum;
        return NULL;
}
```

12.35 Implement a concurrent version of the TINY Web server based on processes. It should create a new child process for each new connection request.

```
while (1) {
        clientlen = sizeof(clientaddr);
        connfd = Accept(
                listenfd, (SA *)&clientaddr, &clientlen);
        if (Fork() == 0) {
                Close(listenfd);
                Getnameinfo(
                        (SA *) &clientaddr, clientlen,
                        hostname, MAXLINE, port,
                        MAXLINE, 0);
                doit(connfd);
                Close(connfd);
                exit(0);
        }
        Close(connfd);
}
```

12.36 Implement a concurrent version of the TINY Web server based on I/O multiplexing.

```
fd_set read_set, ready_set;
FD_ZERO(&read_set);
FD_SET(listenfd, &read_set);

while (1) {
        ready_set = read_set;
        Select(listenfd+1, &ready_set, NULL, NULL, NULL);
        if (FD_ISSET(listenfd, &ready_set)) {
                clientlen = sizeof(clientaddr);
                connfd = Accept(
                        listenfd, (SA *)&clientaddr,
                        &clientlen);
                Getnameinfo(
                        (SA *) &clientaddr, clientlen,
                        hostname, MAXLINE, port,
                        MAXLINE, 0);
                doit(connfd);
                Close(connfd);
        }
}
```

12.37 Implement a concurrent version of the TINY Web server based on threads.

```c
int *connfdp;
pthread_t tid;

while (1) {
        clientlen = sizeof(clientaddr);
        Getnameinfo(
                (SA *) &clientaddr, clientlen, hostname,
                MAXLINE, port, MAXLINE, 0);
        connfdp = malloc(sizeof(int));
        *connfdp = Accept(
                listenfd, (SA *)&clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
}

void *thread(void *vargp)
{
        int connfd = *((int *)vargp);
        pthread_detach(pthread_self());
        free(vargp);
        doit(connfd);
        Close(connfd);
        return NULL;
}
```