Sergio Garcia Tapia
Computer Systems: A Programmer's Perspective, by Bryant and O'Hallaron
Chapter 12: Concurrent Programming
May 28, 2024

Practice Problems

Exercise 12.1. After the parent closes the connected descriptor in line 33 of the concurrent server in Figure 12.5, the child is still able to communicate with the client using its copy of the descriptor. Why?

Solution: When the parent forks a child to handle the request for the client, an identical but independent file descriptor table is created for it. The file descriptors reference the same entries in the table of open descriptions (file). As a result, the reference count of each connected descriptor that existed before the fork increases to 2. The call to `close()` in the parent reduces the reference count to 1, and since it is nonzero, it remains open. Therefore the connection is still active until the child terminates, because at that point the kernel closes any remaining descriptors held open by the child.

Exercise 12.2. If we were to delete line 30 of Figure 12.5, which closes the connected descriptor, the code would still be correct, in the sense that there would be no memory leak. Why?

Solution: When the child process terminates after calling `exit()`, the kernel releases all associated resources, and as part of the clean up it closes all file descriptors previously held open by the child.

Exercise 12.3. In Linux systems, typing Ctrl+D indicates EOF on standard input. What happens if you type Ctrl+D to the program in Figure 12.6 while it is blocked in the call to `select`?

Solution: The call to `select` blocks until a file descriptor in the provided read set is *ready*, meaning that a request to read at least 1 byte from it can be fulfilled without blocking. When the `read()` system call encounters `EOF`, it immediately returns 0. Therefore, file descriptor 0 would become ready, allowing `select` to unblock and return its value.

Exercise 12.4. In the server in Figure 12.8, we are careful to reinitialize the `pool.ready_set` variable immediately before every call to `select`. Why?

Solution: Both `add_client()` and `check_clients()` modify the read set to which `pool.ready_set` was previously initialized. On the hand, we do not want to block waiting for I/O on connected descriptors removed by `check_clients()`. On the other hand, we want to ensure we detect I/O on connected descriptors newly added by `add_client`, and we inform our intent to wait for these descriptors by passing them to `Select()`.

Exercise 12.5. In the process-based server in Figure 12.5, we were careful to close the connected descriptor in two places: the parent process and the child process. However, in the threads-based server in Figure 12.14, we only closed the connected descriptors in one place: the peer thread. Why?

Solution: The main thread and its peer thread share the file descriptor table because the creation of a new thread does not duplicate the file descriptor table. Thus the reference count to the connected descriptor remains at 1. If the main thread closed the connected descriptor, this would terminate the connection with the client before the peer thread could service it. Even if the peer thread were able to service the client and close its descriptor before the main thread was scheduled to run, the main thread would still attempt to close the descriptor. Closing a file descriptor that is already closed is an error.

Exercise 12.6. (a) Using the analysis from Section 12.4, fill each entry in the following table with "Yes" or "No" for the example program in Figure 12.15. In the first column, the notation $v.t$ denotes an instance of variable v residing on the local stack for thread t, where t is either `m` (main thread), `p0` (peer thread 0), or `p1` (peer thread 1).

	Referenced by		
Variable Instance	main thread?	peer thread 0?	peer thread 1?
`ptr`			
`cnt`			
`i.m`			
`msgs.m`			
`myid.p0`			
`myid.p1`			

(b) Given the analysis in part A, which of the variables `prt`, `cnt`, `i`, `msgs`, and `myid` are shared?

Solution:

(a) The `ptr` variable is referenced by the main thread; it is assigned to main's automatic `msgs` variable. It is also referenced by both peer threads to access their own individualized message in main's `msgs` variable through `ptr`.

The `cnt` variable is a local static variable accessible only by invocations of the `thread()` routine. This means that both of the peer threads can access it, and they indeed do on line 26.

The `i.m` variable is a local automatic variable that is only referenced by the main thread.

The `msgs.m` variable is defined in `main`, and it is indirectly referenced in both peer threads through the global `ptr` variable.

The local automatic variable `myid` in the `thread()` procedure run by the peer threads is only referenced by each respective thread where it is defined.

	Referenced by		
Variable Instance	main thread?	peer thread 0?	peer thread 1?
`ptr`	Yes	Yes	Yes
`cnt`	No	Yes	Yes
`i.m`	Yes	No	No
`msgs.m`	Yes	Yes	Yes
`myid.p0`	No	Yes	No
`myid.p1`	No	No	Yes

(b) A variable is shared if and only if one of its instances is referenced by more than one thread. Based on the table above, the variables `ptr`, `cnt`, and `msgs` are all shared.

Exercise 12.7. Complete the table for hte following instruction ordering of `badcnt.c`:

Step	Thread	Instr.	`%rdx`$_1$	`%rdx`$_2$	`cnt`
1	1	H_1	—	—	0
2	1	L_1	_____	_____	_____
3	2	H_2	_____	_____	_____
4	2	L_2	_____	_____	_____
5	2	U_2	_____	_____	_____
6	2	S_2	_____	_____	_____
7	1	U_1	_____	_____	_____
8	1	S_1	_____	_____	_____
9	1	T_1	_____	_____	_____
10	2	T_2	_____	_____	_____

Does this ordering result in a correct value for `cnt`?

Solution:

Step	Thread	Instr.	`%rdx`$_1$	`%rdx`$_2$	`cnt`
1	1	H_1	—	—	0
2	1	L_1	0	—	0
3	2	H_2	—	—	0
4	2	L_2	—	0	0
5	2	U_2	—	1	0
6	2	S_2	—	1	1
7	1	U_1	1	—	1
8	1	S_1	1	—	1
9	1	T_1	1	—	1
10	2	T_2	—	1	1

It does not compute the correct value for `cnt`.

Exercise 12.8. Using the progress graph in Figure 12.21, classify the following trajectories as *safe* or *unsafe*.

(a) H_1, L_1, U_1, S_1, H_2, L_2, U_2, S_2, T_2, T_1

(b) H_2, L_2, H_1, L_1, U_1, S_1, T_1, U_2, S_2, T_2

(c) H_1, H_2, L_2, U_2, S_2, L_1, U_1, S_1, T_1, T_2

Solution:

1. It is safe because the trajectory abuts the critical section.
2. It is unsafe because the trajectory enters the critical region.
3. It is safe because the trajectory abuts the upper perimeter of the critical region but does not enter it.

Exercise 12.9. Let p denote the number of producers, c the number of consumers, and n the buffer size in units of items. For each of the following scenarios, indicate whether the mutex semaphore in `sbuf_insert` and `sbuf_remove` is necessary or not.

(a) $p = 1$, $c = 1$, $n > 1$

(b) $p = 1$, $c = 1$, $n = 1$

(c) $p > 1$, $c > 1$, $n = 1$

Solution:

(a) Yes. The mutex semaphore provides mutually exclusive access to the bounded buffer. It prevents the producer from adding items to the buffer if it becomes full, and the consumer from removing items when it is empty.

(b) No. When the buffer is initialized, it has a slot available, so `sbuf_insert` can announce that it is available. This allows the consumer to operate, and upon finishing announce that a slot is available. The buffer would only be necessary to keep track of pending items, of which there will never be any.

(c) No, for the same reason as (b). Note that the semaphore invariant guarantees that a properly initialized negative value will not attain a negative value. Thus, once any of the producers indicate that there is an item available, only one consumer will be woken to consume it.

Exercise 12.10. The solution to the first readers-writers problem in Figure 12.26 gives priority to readers, but this priority is weak in the sense that a writer leaving its critical section might restart a waiting writer instead of a waiting reader. Describe a scenario where this weak priority would allow a collection of writers to starve a reader.

Solution: If we have two writers, it's possible that the writer that currently holds the mutex will do so until another writer blocks on the P operation waiting for the mutex. Then the writer holding the mutex releases it while the writer blocked waiting for the mutex locks it. By passing it back and forth, the reader is starved. According to the solution provided by the authors, this may happen if the threading implementation uses a LIFO stack.

Exercise 12.11. Fill in the blanks for the parallel program in the following table. Assume strong scaling.

Threads (t)	1	2	4
Cores (p)	1	2	4
Running time (T_p)	12	8	6
Speedup (S_p)	_____	1.5	_____
Efficiency (E_p)	100%	_____	50%

Solution: The *strong scaling* formulation measures the *speedup* of a parallel program as

$$S_p = \frac{T_1}{T_p}$$

where p is the number of processor cores and T_k is the running time on k cores. Meanwhile, the *efficiency* is percentage in the range $(0, 100]$ defined as

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}$$

Efficiency is a measure of the overhead due to parallelization, where high efficiency means that more time is spent doing useful work and less time synchronizing and communicating.

Threads (t)	1	2	4
Cores (p)	1	2	4
Running time (T_p)	12	8	6
Speedup (S_p)	1	1.5	2
Efficiency (E_p)	100%	75%	50%

Exercise 12.12. The `ctime_ts` function in Figure 12.38 is thread-safe but nto reentrant. Explain.

Solution: If multiple threads execute the function, then they all share the static variable that is local to the thread-unsafe `ctime()` function. The `ctime_ts` function is thread-safe because it uses a mutex semaphore to protect this share variable. However, reentrant functions are those that do not reference *any* shared data when called by multiple threads. Since `ctime_ts` fails to satisfy this condition on account of the `sharedp` variable, it is non-reentrant.

Exercise 12.13. In Figure 12.43, we might be tempted to free the allocated memory block immediately after line 14 in the main thread, instead of freeing it in the peer thread. But that would be a bad idea. Why?

Solution: This would create a race between main and a peer thread. If the peer thread is scheduled first and runs through completion, then the program completes normally. However, if the main thread frees the memory for the heap-allocated variable that it passed to the peer thread, then the peer thread will reference an invalid memory location, causing the kernel to generate a `SIGSEGV`.

Exercise 12.14. (a) In Figure 12.43, we eliminated the race by allocating a separate block for each integer ID. Outline a different approach that does not call the `malloc` or `free` functions.

(b) What are the advantages and disadvantages of this approach?

Solution:

(a) The main thread can create a local automatic array. On each loop iteration, it assigns the thread ID, and then it passes a pointer to the memory location in the array of this index.

In the book, the authors suggest instead passing a pointer to integer `i` directly, rather than a pointer to `i`. Then in the peer thread, the argument is cast back to an `int`.

(b) Once advantage is that we limit the overhead of the function calls, including the underlying system calls needed to adjust the system break of the process. It also ensures that memory is automatically managed by the stack discipline. Moreover, it may even enjoy more spatial locality. The disadvantage is that there is a risk that if the main thread exits before any peer thread, then its stack frame is deallocated, leaving the peer threads pointing to possibly invalid memory locations. It also requires us to know ahead of time the number of threads that we will be creating.

In the book, following the suggestion from the authors that I wrote in part (a), the advantage is again that the overhead of the function calls is reduced. Meanwhile, the disadvantage is that its correctness is dependent the pointers being at least as large as `int`s, which may not be true for legacy or future systems.

Exercise 12.15. Consider the following program, which attempts to use a pair of semaphores for mutual exclusion:

```
Initially: s = 1, t = 0.

Thread 1:          Thread 2:
    P(s);              P(s);
    V(s);              V(s);
    P(t);              P(t);
    V(t);              V(t);
```

(a) Draw the progress graph for this program.

(b) Does it always deadlock?

(c) If so, what simple change to the initial semaphore values will eliminate the potential for deadlock?

(d) Draw the progress graph for the resulting deadlock-free program.