Computing platforms (Spring 2025)
week 6

Exercise 1

Readers/Writers with priorities. The lecture slides give an example of solving the Readers/Writers problem with locks and condition variables. That solution is inefficient in the sense that when both writers and readers are waiting when a writer finishes, they will fight over who will go next. The python code "readers-writers.py" gives another solution that solves the problem. Download the code separately from Moodle.

(a) Can a writer starve in this solution? Can a reader starve in this solution?

Writer can starve. Once writer finishes all readers are allowed to proceed in concurrent manner. This race can preempt waiting writer and possibly starve writer.

Reader cannot starve. When writer finish all readers can proceed simultaneously.

(b) Who has higher priority, readers or writers? Why?

Readers have higher priority When writer finishes all readers proceed together, this give advantage to readers.

(c) Modify the code so that writers have priority.

```
import threading

# Keep track of currently reading threads
readers=set()
# Keep track of currently writing threads
writers=set()
# Keep track of the number of readers waiting to read
waitingreaders=0
# Keep track of the number of writers waiting to write
waitingwriters=0
# Lock for protecting the shared variables
lock = threading.Lock()
# Condition variable where readers wait for their turn to read
oktoread = threading.Condition(lock)
# Condition variable where writers wait for their turn to write
oktowrite = threading.Condition(lock)

# Function executed by the readers
def reader():
    global readers
    global writers
    global waitingreaders
```

```
    global waitingwriters
    # Retrieve the name of this thread
    name = threading.current_thread().name
    # Each reader will read three times
    for _ in range(3):
        lock.acquire()
        # If there are currently writers readers needs to wait
        while len(writers) > 0 or waitingwriters > 0 :
            waitingreaders += 1
            oktoread.wait()
            waitingreaders -= 1
        # Now there are no writers
        readers.add(name)
        print(name + ": Starts reading")
        print("Current readers: " + str(readers) + " Current writers: " + str(writers))
        lock.release()
        print(name + ": Reading...")
        lock.acquire()
        print(name + ": Finishes reading")
        readers.remove(name)
        # If there are no readers left, let the writers start writing
        if len(readers) == 0:
            oktowrite.notify()
        lock.release()

# Function executed by the writers
def writer():
    global readers
    global writers
    global waitingreaders
    global waitingwriters
    # Retrieve the name of this thread
    name = threading.current_thread().name
    # Each writer writes three times
    for _ in range(3):
        lock.acquire()
        # Modified for writer priority
        while len(writers) > 0 or len(readers) > 0 :
            waitingwriters += 1
            oktowrite.wait()
            waitingwriters -= 1
        # Now there are no current readers/writers
        writers.add(name)
        print(name + ": Starts writing")
        print("Current readers: " + str(readers) + " Current writers: " + str(writers))
        lock.release()
        print(name + ": Writing...")
        lock.acquire()
        print(name + ": Finishes writing")
        writers.remove(name)
        # Modified notification order: notify waiting writers first, then readers
        if waitingwriters > 0 :
            oktowrite.notify()
        elif waitingreaders > 0:
            oktoread.notify_all()
        lock.release()
```

```
# Create the reader and writer threads
readerThreads = [threading.Thread(target=reader, name = "R" + str(i)) for i in range(10)]
writerThreads = [threading.Thread(target=writer, name = "W" + str(i)) for i in range(10)]

# Start the reader and writer threads
for i in range(10):
    readerThreads[i].start()
    writerThreads[i].start()
```

(d) Can a reader starve in your solution? Can a writer starve in your solution?

Reader: yes, can starve. if writers keep on arriving continously readers never get to execute.

Writer: no, cant́ starve. Writers have priority, if there is writer waiting readers will be blocked.

Exercise 2

Deadlock. (Problem 6.6 from [Stallings 2018], modified to Python) In the code below, three threads are competing for six resources labeled A to F.

```
#thread 1       #thread 2       #thread 3
while True:     while True:     while True:
    get(A)          get(D)          get(C)
    get(B)          get(E)          get(F)
    get(C)          get(B)          get(D)
    # critical region: # critical region: # critical region:
    # use A, B, C   # use D, E, B   # use C, F, D
    release(A)      release(D)      release(C)
    release(B)      release(E)      release(F)
    release(C)      release(B)      release(D)
```

(a) Using a resource allocation graph (Figures 6.5 and 6.6 in [Stallings 2018], also in lecture slides), show the possibility of deadlock in this implementation.

we have resource allocation graph with two types of vertices:

- **Threads (T1, T2, T3)**.
- **Resources (A, B, C, D, E, F)**.

then we draw:

- T $\rightarrow$ R if thread T is requesting resource R.
- R $\rightarrow$ T if resource R is currently allocated to thread T.

Possible interleaving that leads to deadlock:

(a) **T1** has successfully acquired $\{A, B\}$ and is requesting C.
(b) **T2** has successfully acquired $\{D, E\}$ and is requesting B.
(c) **T3** has successfully acquired $\{C, F\}$ and is requesting D.

From here we have:

$$T1 \rightarrow C, \quad C \rightarrow T3, \quad T3 \rightarrow D, \quad D \rightarrow T2, \quad T2 \rightarrow B, \quad B \rightarrow T1.$$

E.g. a cycle:

$$T1 \rightarrow C \rightarrow T3 \rightarrow D \rightarrow T2 \rightarrow B \rightarrow T1,$$

Deadlock situation where each thread is holding resource next thread want.

(b) Modify the order of some of the get requests to prevent the possiblity of any deadlock. You cannot move requests across threads, only change the order inside each thread. Use a resource allocation graph to justify your answer.

We can enforce **global ordering** of resource acquisition. Suppose we impose alphabetical order $A < B < C < D < E < F$. Threads must request its resources in ascending order and release them in reverse order. A possible reordering is:

- **Thread 1**: `get(A)`, `get(B)`, `get(C)`
- **Thread 2**: `get(B)`, `get(D)`, `get(E)`
- **Thread 3**: `get(C)`, `get(D)`, `get(F)`

No circular wait can form because there is no way to construct a cycle if every thread requests resources according to the same global ordering.

Revised Resource Allocation Graph

- **T1** edges always proceed $T1 \to A \to B \to C$.
- **T2** edges always proceed $T2 \to B \to D \to E$.
- **T3** edges always proceed $T3 \to C \to D \to F$.

No thread requests resource with lower label than any resource it already holds hence we avoid circular-wait condition.

Exercise 3

SSDs. In this exercise, we use a Python simulator from OSTEP. You find the code at https://github.com/remzi-arpacidusseau/ostep-homework/blob/master/file-ssd. Download the Python code ssd.py and read the README of the simulator. Answer the homework questions 1-5 from OSTEP Chapter 44. You find the homework questions at the end of the chapter at https://pages.cs.wisc.edu/ remzi/OSTEP/file-ssd.pdf. For question 1 running with seed 1 suffices.

1. The homework will mostly focus on the log-structured SSD, which is simulated with the "-T log" flag. We'll use the other types of SSDs for comparison. First, run with flags -T log -s 1 -n 10 -q. Can you figure out which operations took place? Use -c to check your answers (or just use -C instead of -q -c). Use different values of -s to generate different random workloads.

```
write(12, u)
write(32, M)
read(32) -> M
write(38, 0)
write(36, e)
trim(36)
read(32) -> M
trim(32)
read(12) -> u
read(12) -> u
```

2. Now just show the commands and see if you can figure out the intermediate states of the Flash. Run with flags -T log -s 2 -n 10 -C to show each command. Now, determine the state of the Flash between each command; use -F to show the states and see if you were right. Use different random seeds to test your burgeoning expertise.

```
                              State iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
cmd   0:: write(36, F) -> success State vEEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
cmd   1:: write(29, 9) -> success State vvEEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
cmd   2:: write(19, I) -> success State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
cmd   3:: trim(19) -> success     State vvvEEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
cmd   4:: write(22, g) -> success State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
cmd   5:: read(29) -> 9           State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
cmd   6:: read(22) -> g           State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
cmd   7:: write(28, e) -> success State vvvvvEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
cmd   8:: read(36) -> F           State vvvvvEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
cmd   9:: write(49, F) -> success State vvvvvvEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
```

3. Let's make this problem ever so slightly more interesting by adding the -r 20 flag. What differences does this cause in the commands? Use -c again to check your answers.

```
cmd   0:: read(41) -> fail: uninitialized read
cmd   1:: write(33, j) -> success
cmd   2:: write(30, A) -> success
cmd   3:: read(33) -> j
cmd   4:: write(49, W) -> success
cmd   5:: write(22, g) -> success
cmd   6:: read(23) -> fail: uninitialized read
cmd   7:: read(22) -> g
cmd   8:: write(28, e) -> success
cmd   9:: read(33) -> j
```

We see failed commands.

4. Performance is determined by the number of erases, programs, and reads (we assume here that trims are free). Run the same workload again as above, but without showing any intermediate states (e.g., -T log -s 1 -n 10). Can you estimate how long this workload will take to complete? (default erase time is 1000 microseconds, program time is 40, and read time is 10) Use the -S flag to check your answer. You can also change the erase, program, and read times with the -E, -W, -R flags.

```
FTL   (empty)
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data
Live
```

```
FTL    12:  0  38:  2
Block 0          1          2          3          4          5          6
Page  0000000000 1111111111 2222222222 3333333333 4444444444 5555555555 6666666666
      0123456789 0123456789 0123456789 0123456789 0123456789 0123456789 0123456789
State vvvvEEEEEE iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii iiiiiiiiii
Data  uMOe
Live  + +

Physical Operations Per Block
Erases   1          0          0          0          0          0          0          Sum: 1
Writes   4          0          0          0          0          0          0          Sum: 4
Reads    4          0          0          0          0          0          0          Sum: 4

Logical Operation Sums
  Write count 4 (0 failed)
  Read count  4 (0 failed)
  Trim count  2 (0 failed)

Times
  Erase time 1000.00
  Write time 160.00
  Read time  40.00
  Total time 1200.00
```

5. Now, compare performance of the log-structured approach and the (very bad) direct approach (-T direct instead of -T log). First, estimate how you think the direct approach will perform, then check your answer with the -S flag. In general, how much better will the log-structured approach perform than the direct one?

Logical mapped:

```
Times
Erase time 1000.00
Write time 160.00
Read time  40.00
Total time 1200.00
```

Direct mapped:

```
Times
Erase time 4000.00
Write time 280.00
Read time  70.00
Total time 4350.00
```

Exercise 4

UNIX file management. Read Chapter 12.8 [Stalling, 2018, pp. 580-585] and answer the following questions. Consider the organization of UNIX file as represented by the inode (see Figure 12.15, also provided below). Assume there are 12 direct block pointers, and a singly, doubly, and triply indirect pointer to each inode. Assume further that the system block size and the disk sector size are both 8 KB, and the disk block pointer is 32 bits, with 8 bits to identify the physical disk and 24 bits to identify the physical block.

(a) What is the maximum file size supported by this system?

- Direct: 12 blocks * 8192B = 98304 B
- Singly indirect: 2048 pointers * 8192B = 16777216 B (16GB)
- Doubly indirect: 2048^2 * 8192 = 4194304 * 8192 $\approx$ 34359738368 B (34GB)
- Triply indirect: 2048^3 * 8192 = 8589934592 * 8192 $\approx$ 70368744177664 B (70TB)

(b) What is the maximum system partition (i.e. physical disk) supported by this system?

- With 24 bits for physical block number:
$$2^{24} = 16777216 \text{ blocks per disk}$$
- Each block is 8192 B:
$$16777216 \times 8192 = 137438953472 \text{ B} = 2^{24+13} = 2^{37} \text{ B}$$
- maximum partition size:
$$2^{37} \text{ B} \approx 137.44 \text{ GB}$$

(c) Assuming no information other than the the file inode is already on the main memory, how many disk accesses are required to access the byte in position 13 423 956.

- block index:
$$\text{Block index} = \left\lfloor \frac{13\,423\,956}{8192} \right\rfloor \approx 1638$$
- Mapping:
  - Direct pointers cover blocks 0 -11
  - The singly indirect block covers blocks 12 to 12 + 2048 - 1 = 2059

  1638 is in singly indirect range:
- Disk accesses needed (inode is already in memory):
  - 1 access to read the singly indirect block.
  - 1 access to read the actual data block.

Exercise 5

Bob wants to analyze the network traffic on his computer. To gather the data, he creates a system that inspects every network packet that goes through his network device. To store the data, he creates a new file for each network packet using a timestamp as a file name and stores the size of the network packet in the file. Soon he has gathered huge amounts of data and he has a huge amount of small files in his system.

(a) What are the disadvantages of this approach?

- Excessive file system overhead.
- Poor performance due to many small file creations and system calls.
- Increased fragmentation and slower file system operations.
- Difficulty managing and backing up a huge number of files.

(b) After running this system for some time, the whole computer system crashed. What could be the reason for this?

- Running out of storage space or inodes due to an enormous number of files.
- Overloaded file system metadata management causing memory exhaustion or degraded system performance.

(c) How should the data be organized instead?

- Store data into larger files. e.g., appending many packets to a single file.
- Organize data by connection or time intervals rather than per packet.
- maybe use database that can efficiently store and index numerous records.