DAY-23 EVENING ASSESSMENT

31)

# at module level  
HOLIDAYS = {  
   # ISO dates  
   "2025-01-26", "2025-08-15", "2025-10-02",  
}  
  
def \_is\_business\_day(d: datetime) -> bool:  
   return d.weekday() < 5 and d.date().isoformat() not in HOLIDAYS  
  
def add\_business\_days(start: datetime, days: int) -> datetime:  
   d = start  
   added = 0  
   while added < days:  
       d += timedelta(days=1)  
       if \_is\_business\_day(d):  
           added += 1  
   return d  
Use in borrow() instead of issue + timedelta(days=m.loan\_days()):  
  
issue = datetime.now()  
due = add\_business\_days(issue, m.loan\_days())  
  
32) Extend Book dataclass:  
  
from dataclasses import dataclass, field  
from typing import List  
  
@dataclass  
class Book:  
   book\_id: str  
   title: str  
   author: str  
   isbn: str  
   available: bool = True  
   last\_modified: str = field(default\_factory=lambda: datetime.now().isoformat())  
   queue: List[str] = field(default\_factory=list)  # member\_id queue  
Portal methods:  
  
def reserve(self, member\_id: str, book\_id: str):  
   with self.\_lock:  
       b = self.books.get(book\_id); m = self.members.get(member\_id)  
       if not b or not m: print("Invalid member/book"); return  
       if b.available:  
           print("Book is available; borrow directly."); return  
       if member\_id in b.queue:  
           print("Already in queue."); return  
       b.queue.append(member\_id)  
       b.last\_modified = datetime.now().isoformat()  
       self.\_save(); self.\_log(f"RESERVE {member\_id} -> {book\_id}")  
       print("Reservation added.")  
  
# modify receive(): auto-issue to next in queue  
def receive(self, member\_id: str, book\_id: str):  
   with self.\_lock:  
       # ...validate + compute fine...  
       m.\_remove\_loan(book\_id)  
       if b.queue:  
           next\_member\_id = b.queue.pop(0)  
           next\_m = self.members.get(next\_member\_id)  
           issue = datetime.now()  
           due = add\_business\_days(issue, next\_m.loan\_days())  
           next\_m.\_add\_loan(book\_id, issue, due)  
           b.available = False  
           msg = f"Auto-issued to {next\_m.name} (queue). Due: {due.date()}"  
       else:  
           b.available = True  
           msg = "Returned to shelf."  
       b.last\_modified = datetime.now().isoformat()  
       self.\_save(); self.\_log(f"RETURN {member\_id} -> {book\_id}")  
       print(msg)  
  
33) When adding a loan:  
m.\_add\_loan(book\_id, issue, due)  # ensure \_add\_loan sets renewed=False  
Update the helper:  
  
def \_add\_loan(self, book\_id: str, issue\_date: datetime, due\_date: datetime):  
   self.\_\_borrowed.append({  
       "book\_id": book\_id,  
       "issue\_date": issue\_date.isoformat(),  
       "due\_date": due\_date.isoformat(),  
       "renewed": False,  
   })  
Add renew():  
  
def renew(self, member\_id: str, book\_id: str):  
   with self.\_lock:  
       m = self.members.get(member\_id); b = self.books.get(book\_id)  
       if not m or not b: print("Invalid member/book"); return  
       if b.queue:  
           print("Cannot renew: reservations pending."); return  
       loan = next((x for x in m.borrowed if x["book\_id"] == book\_id), None)  
       if not loan: print("No active loan."); return  
       if loan.get("renewed"): print("Already renewed once."); return  
       due = datetime.fromisoformat(loan["due\_date"])  
       if datetime.now().date() > due.date():  
           print("Cannot renew overdue books."); return  
       new\_due = add\_business\_days(due, m.loan\_days() // 2)  # e.g., half of original  
       loan["due\_date"] = new\_due.isoformat()  
       loan["renewed"] = True  
       b.last\_modified = datetime.now().isoformat()  
       self.\_save(); self.\_log(f"RENEW {member\_id} -> {book\_id} new\_due={new\_due.date()}")  
       print(f"Renewed. New due: {new\_due.date()}")  
  
34) Track counts whenever borrowing:  
  
from collections import defaultdict  
# in \_\_init\_\_  
self.\_borrow\_counts = defaultdict(int)  
  
# in borrow()  
self.\_borrow\_counts[b.book\_id] += 1  
self.\_log(f"BORROW {member\_id} -> {book\_id} due={due.date()}")  
Monthly report:  
  
def top\_borrowed(self, year: int, month: int, top\_n: int = 10):  
   # parse append-only action log for month scope  
   counts = defaultdict(int)  
   with open(ACTION\_LOG, "r", encoding="utf-8") as f:  
       for line in f:  
           # line format: 2025-08-20T12:34:56 BORROW MID -> BID due=...  
           if " BORROW " in line and f"{year:04d}-{month:02d}-" in line[:10]:  
               book\_id = line.split("->")[1].split()[0]  
               counts[book\_id] += 1  
   ranked = sorted(counts.items(), key=lambda kv: kv[1], reverse=True)[:top\_n]  
   for bid, c in ranked:  
       bk = self.books.get(bid)  
       print(f"{bk.title if bk else bid} — {c} borrows")  
  
35) def iter\_books(self):  
   for b in self.books.values():  
       yield b  
  
# streaming from file (TXT) without loading all:  
def stream\_books\_from\_file(self, path=BOOKS\_FILE\_NG):  
   with open(path, "r", encoding="utf-8") as f:  
       for line in f:  
           if line.strip():  
               yield Book.from\_line(line)  
Use:  
  
for b in portal.iter\_books(): ...  
for b in portal.stream\_books\_from\_file(): ...  
  
36)   
import cProfile, pstats, io  
from nextgen\_library\_portal import LibraryPortal  
  
def workload():  
   p = LibraryPortal()  
   # synthetic hot path  
   for i in range(500):  
       bid = f"B{i}"  
       p.insert\_book(bid, f"Title {i}", "Author", f"ISBN{i}")  
   p.search\_books("title")  
   p.sort\_books("author")  
   return p.available\_books\_count  
  
if \_\_name\_\_ == "\_\_main\_\_":  
   pr = cProfile.Profile()  
   pr.enable()  
   workload()  
   pr.disable()  
   s = io.StringIO()  
   ps = pstats.Stats(pr, stream=s).sort\_stats("tottime")  
   ps.print\_stats(30)  
   print(s.getvalue())  
  
  
37) Cache frequently accessed lookups (lru\_cache)  
Cache a stable lookup, e.g., by ISBN. Include a version token so cache invalidates after mutations:  
  
from functools import lru\_cache  
  
class LibraryPortal:  
   def \_\_init\_\_(self):  
       ...  
       self.\_cache\_version = 0  
  
   def \_bump\_cache(self):  
       self.\_cache\_version += 1  
  
   def \_save(self):  
       # ... your save ...  
       self.\_bump\_cache()  
  
   @lru\_cache(maxsize=2048)  
   def get\_book\_by\_isbn\_cached(self, isbn: str, version: int):  
       # version is unused but part of key to bust cache  
       for b in self.books.values():  
           if b.isbn == isbn:  
               return b.book\_id  
       return None  
  
   def get\_book\_by\_isbn(self, isbn: str):  
       return self.get\_book\_by\_isbn\_cached(isbn, self.\_cache\_version)  
Call get\_book\_by\_isbn("..."); cache is auto-invalidated when \_save() bumps version.  
  
38) processes don’t share in-memory portal; each process re-opens state from disk. Your \_safe\_write + append-only log help tolerate contention (this is a stress simulation, not a transactional DB).  
  
from multiprocessing import Pool  
  
def \_borrow\_worker(args):  
   member\_id, book\_id = args  
   p = LibraryPortal()  # each process loads state  
   try:  
       p.borrow(member\_id, book\_id)  
   except Exception as e:  
       p.\_log(f"MP\_BORROW\_FAIL {member\_id} {book\_id} {e}")  
  
def simulate\_mp\_100(portal: LibraryPortal, book\_id: str):  
   args = [(f"M{i:03d}", book\_id) for i in range(100)]  
   with Pool(processes=8) as pool:  
       pool.map(\_borrow\_worker, args)  
Expect exactly one successful borrow; others should fail (book not available) or get queued if you call reserve() for them.  
  
39) defaultdict and OrderedDict  
  
from collections import defaultdict, OrderedDict  
  
class LibraryPortal:  
   def \_\_init\_\_(self):  
       self.books = OrderedDict()     # preserves insertion order (e.g., for listing)  
       self.members = OrderedDict()  
       self.\_borrow\_counts = defaultdict(int)  # cheap counters  
       ...  
  
40)   
import time, json, random, string  
  
def \_random\_str(n=8):  
   return "".join(random.choices(string.ascii\_letters + string.digits, k=n))  
  
def generate\_books(n=10000):  
   return [Book(f"B{i}", f"Title {\_random\_str(6)}", f"Author {\_random\_str(5)}", f"978{1000000000+i}")  
           for i in range(n)]  
  
def benchmark\_persistence():  
   p = LibraryPortal()  
   # populate  
   for b in generate\_books(10000):  
       p.books[b.book\_id] = b  
   # TXT  
   t0 = time.perf\_counter(); p.\_save(); t1 = time.perf\_counter()  
   txt\_time = t1 - t0  
   # JSON  
   t0 = time.perf\_counter(); p.\_save\_json(); t1 = time.perf\_counter()  
   json\_time = t1 - t0  
   print(f"TXT save: {txt\_time:.3f}s | JSON save: {json\_time:.3f}s")