DAY 23 : EVENING ASSESSMENT

# 31. Add a holiday calendar so due dates skip weekends/holidays

HOLIDAYS = {"2025-01-01", "2025-08-15"}  
def compute\_due\_date\_with\_holidays(self, borrow\_date):  
 if hasattr(borrow\_date, 'date'):  
 due = borrow\_date + timedelta(days=14)  
 else:  
 due = datetime.combine(borrow\_date, datetime.min.time()) + timedelta(days=14)  
 while due.weekday() >= 5 or due.strftime("%Y-%m-%d") in HOLIDAYS:  
 due += timedelta(days=1)  
 return due

# 32. Allow books to be reserved: queue next member if unavailable

from collections import deque  
from threading import Lock  
  
class Book:  
 def \_\_init\_\_(self, book\_id, title, author, isbn, available=True):  
 self.book\_id = book\_id  
 self.title = title  
 self.author = author  
 self.isbn = isbn  
 self.available = available  
 self.reservation\_queue = deque()  
  
class LibrarySystem:  
 \_reserve\_lock = Lock()  
 def enqueue\_reservation(self, book\_id, user\_id):  
 with self.\_reserve\_lock:  
 book = self.books.get(book\_id)  
 if not book:  
 raise LookupError("Book not found")  
 if not book.available:  
 book.reservation\_queue.append(user\_id)  
 return f"Book reserved for {user\_id}"  
 return "Book is currently available; you can borrow it now."  
  
 def \_notify\_next\_reserver(self, book):  
 if book.reservation\_queue:  
 next\_user = book.reservation\_queue.popleft()

# 33. Implement a renewal system: members can extend due date only once

def renew\_loan\_once(self, book\_id, user\_id, days=7):  
 for t in self.transactions:  
 if t.get("book") == book\_id and t.get("user") == user\_id and "return\_date" not in t:  
 if t.get("renewed"):  
 return "Already renewed once"  
 t["due\_date"] = (datetime.fromisoformat(t["due\_date"]) + timedelta(days=days)).isoformat()  
 t["renewed"] = True  
 self.\_persist\_state()  
 return "Renewed successfully"  
 return "No active loan found for this user/book"

# 34. Track and print a monthly report of top borrowed books

from collections import Counter  
  
def generate\_monthly\_top\_books\_report(self, year, month, top\_n=5):  
 books\_borrowed = [  
 t["book"] for t in self.transactions  
 if "date" in t and datetime.fromisoformat(t["date"]).year == year  
 and datetime.fromisoformat(t["date"]).month == month  
 ]  
 counter = Counter(books\_borrowed)  
 return counter.most\_common(top\_n)  
  
def format\_top\_books\_report(self, top\_list):  
 lines = [f"{rank}. {book\_id} — {count} borrows" for rank, (book\_id, count) in enumerate(top\_list, 1)]  
 return "\n".join(lines)

# 35. Use generators to lazily iterate through all books

def iter\_all\_books(self):  
 for book in self.books.values():  
 yield book

# 36. Profile the system using cProfile and identify bottlenecks

import cProfile, pstats, io  
  
def profile\_system\_run(self, entry\_callable, sort\_by="cumulative"):  
 pr = cProfile.Profile()  
 pr.enable()  
 entry\_callable()  
 pr.disable()  
 s = io.StringIO()  
 ps = pstats.Stats(pr, stream=s).sort\_stats(sort\_by)  
 ps.print\_stats(50) # top 50 lines  
 return s.getvalue()

# 37. Cache frequently accessed books using functools.lru\_cache

from functools import lru\_cache  
  
def \_books\_version(self):  
 return getattr(self, "\_books\_ver", 0)  
  
def \_bump\_books\_version(self):  
 self.\_books\_ver = self.\_books\_version() + 1  
  
@lru\_cache(maxsize=256)  
def \_cached\_get\_book(self, version, book\_id):  
 return self.books.get(book\_id)  
  
def cached\_get\_book(self, book\_id):  
 return self.\_cached\_get\_book(self.\_books\_version(), book\_id)

# 38. Use multiprocessing to simulate 100 members borrowing simultaneously

from multiprocessing import Pool, get\_context  
from functools import partial  
  
def \_worker\_borrow(args):  
 lib, user\_id, book\_id = args  
 try:  
 lib.borrow\_book(book\_id, user\_id)  
 return (user\_id, "OK")  
 except Exception as e:  
 return (user\_id, f"ERR: {e}")  
  
def simulate\_parallel\_borrow(self, book\_id="B1", users=100, processes=8):  
 ctx = get\_context("spawn")  
 with ctx.Pool(processes) as pool:  
 args\_iter = ((self, f"U{i}", book\_id) for i in range(users))  
 results = pool.map(\_worker\_borrow, args\_iter)  
 return results

# 39. Replace dicts with defaultdict/OrderedDict where applicable

from collections import defaultdict, OrderedDict  
  
def init\_collection\_structures(self):  
 if not isinstance(self.books, OrderedDict):  
 self.books = OrderedDict(self.books)  
 if not isinstance(self.transactions, defaultdict):  
 dd = defaultdict(list)  
 if isinstance(self.transactions, list):  
 for t in self.transactions:  
 dd[t.get("user")].append(t)  
 elif isinstance(self.transactions, dict):  
 for k, v in self.transactions.items():  
 dd[k].extend(v if isinstance(v, list) else [v])  
 self.transactions = dd

# 40. Benchmark file vs JSON persistence with large data

import csv, json, time, tempfile, os  
  
def benchmark\_persistence(self, n=10000):  
   
 books = [  
 {"book\_id": f"B{i}", "title": f"T{i}", "author": "A", "isbn": f"ISBN{i}", "available": True}  
 for i in range(n)  
 ]  
 timings = {}  
 tj0 = time.perf\_counter()  
 with tempfile.NamedTemporaryFile(delete=False, suffix=".json") as f:  
 json.dump(books, f)  
 json\_path = f.name  
 timings["json\_write\_s"] = time.perf\_counter() - tj0  
  
 tj1 = time.perf\_counter()  
 with open(json\_path, "r") as f:  
 \_ = json.load(f)  
 timings["json\_read\_s"] = time.perf\_counter() - tj1  
 os.remove(json\_path)  
  
 tc0 = time.perf\_counter()  
 with tempfile.NamedTemporaryFile(delete=False, suffix=".csv", mode="w", newline="") as f:  
 w = csv.writer(f)  
 w.writerow(["book\_id","title","author","isbn","available"])  
 for b in books:  
 w.writerow([b["book\_id"], b["title"], b["author"], b["isbn"], b["available"]])  
 csv\_path = f.name  
 timings["csv\_write\_s"] = time.perf\_counter() - tc0  
  
 tc1 = time.perf\_counter()  
 read\_csv = []  
 with open(csv\_path, newline="") as f:  
 r = csv.DictReader(f)  
 for row in r:  
 row["available"] = (row["available"] == "True")  
 read\_csv.append(row)  
 timings["csv\_read\_s"] = time.perf\_counter() - tc1  
 os.remove(csv\_path)  
  
 return timings