Introduction

In student dormitories, laundry rooms are vital shared spaces - but managing them can quickly turn into chaos. Picture a busy Sunday evening: a handful of students, each armed with a basket of clothes, jostle for the limited washers and dryers. To prevent overcrowding, the dorm has a system: only a few students can enter the laundry room at once, using physical keys handed out at the entrance. Inside, the unspoken rule is clear: first wash your clothes, then dry them.

But laundry isn't a continuous operation - machines take time to run. During these idle moments, students often wander off to take a break, grabbing a snack or chatting with friends, before returning to claim their spot. With limited keys, shared appliances, and students intermittently leaving the room, it becomes a delicate balance to keep the laundry flowing without blocking others. Mismanaging this flow could easily cause bottlenecks, long waits, or even deadlocks where no one can make progress.

To model and solve this, we define the following rules for the system:

Rules

- **Limited Keys**: Entry to the laundry room is controlled by a fixed number of keys (e.g., 4 keys total).
- **Mandatory Entry**: A student must hold a key and be inside the room to use any appliance.
- **Ordered Usage**: Students must first use a washing machine, then a dryer - no skipping or reordering.

- **Resting Behavior**: While a machine is running, a student has a *70%* chance of temporarily leaving the room to rest.
- **Reentry Requirement**: After resting, the student must wait for a key again to re-enter and continue their laundry.

Problem Statement

In the shared laundromat system, students must carefully coordinate their actions to avoid conflicts over limited resources. Access to the room itself is restricted by a finite number of keys, and students must sequentially use a washing machine followed by a dryer. However, unpredictability is introduced because students may choose to leave the room to rest while their laundry is running, creating dynamic changes in room occupancy.

Without careful synchronization, several issues can arise:

- **Deadlocks**: If students hold on to machines or keys while resting, others may be indefinitely blocked from progressing through their laundry tasks. Example: If as many students as there are keys leave on a break and during that time just as many students enter the laundry then there is a deadlock - **N** students hold an appliance and **N** students hold a key. This means that the first bunch of students are locked outside and cannot enter, forcing the students inside to wait for available washing machines and dryers.
- **Fairness Issues (starvation)**: Some students might repeatedly gain access while others are forced to wait too long.
- **Underutilization**: Machines may sit idle because keys or students are stuck waiting elsewhere in the system.

The core challenge is designing a mechanism that ensures students can wash and dry their clothes in order, can rest if needed, and yet avoid creating system-wide stalls or unfair bottlenecks.

Design and Approach

To manage student behavior in the laundromat without causing deadlocks, we explored several synchronization strategies. Early prototypes ran into deadlocks when students read stale values of shared counters or when too many students rested at the same time, blocking new entries. To address these issues, we implemented a system based on the following principles:

Keys

A semaphore limits the number of students allowed into the room at any time. They are acquired at the beginning of the loop and released either at the very end of the process or when a students goes for a break.

Resting Control

While waiting for machines, students have a 70% chance to rest, but the number of resting students is restricted to at most *(n - 1)* to prevent stalling the system. This works by always keeping at least one student in the system such that he can keep the resources moving by releasing appliances and keys. Since he is not allowed to rest he has to eventually release his resources. This is certain, because the resources of washing machine and dryer are sequential and cannot raise a deadlock.

Mutex

All counters and shared data is protected by a mutex which prevents different threads from reading stale data or overriding each other. Example protected data is the amount of resting people that we use to enforce the previous point.

Semaphores

Semaphores are used to implement the concept of a finite set of resources such as washing machines, dryers and physical keys.

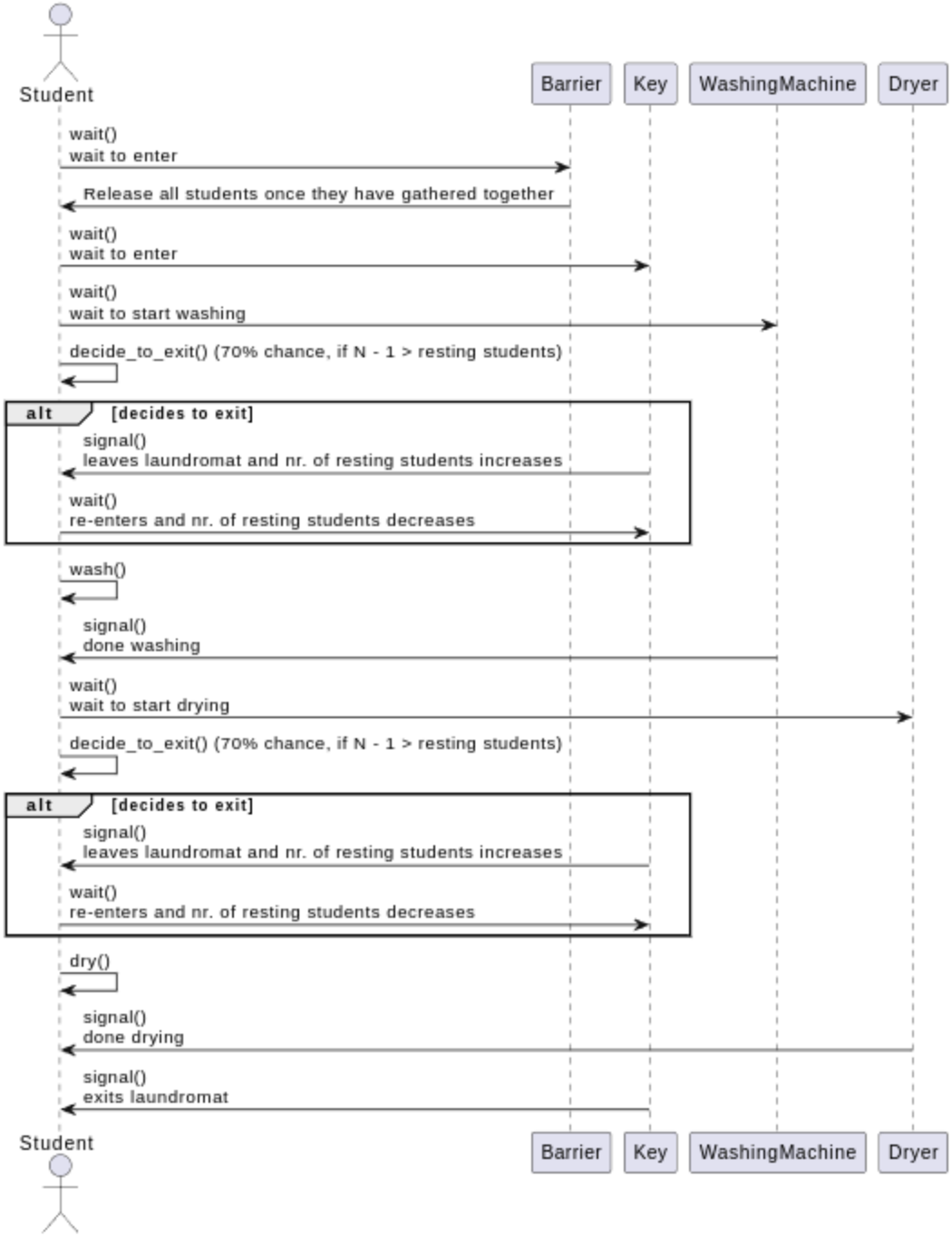
Through these strategies, we ensure that students can leave and re-enter without causing deadlock or resource starvation, maintaining the system’s liveness and fairness.

Design

The UML sequence diagram we developed captures a typical lifecycle of a student in the concurrent laundromat simulation. It illustrates the critical steps in the system: waiting at the barrier, entering the laundromat with a key, using the washing machine and dryer in order, and optionally taking a rest in between. Each student thread is represented by a lifeline, and a `barrier` and semaphores such as `keys`, `washing_machine`, and `dryer` appear as distinct objects that synchronize the students’ actions.

What makes this diagram especially relevant is its focus on coordination rather than just execution. The `barrier.wait()` call - placed before `keys.wait()` - ensures that all students enter the system fairly by requiring them to wait until a full group (equal to the number of students) is ready. This mechanism avoids starvation and ensures no student repeatedly gets to go before all others have completed the same amount of "trips". Once all students have reached the barrier, they are simultaneously released, creating fair competition for available keys.

In this way, the sequence diagram reflects both the logical flow of operations and the synchronization discipline imposed to avoid race conditions, deadlocks, or unfairness. Each interaction is clearly timestamped, allowing viewers to see the temporal dependencies between actions like entering, resting, and waiting on semaphores. Furthermore, by showing how the barrier batches students before entering the laundromat, the diagram directly supports our claim of fairness and controlled access to the system’s limited resources.



Specific numbers and constraints

The amounts of students, washing machines and dryers can vary freely. During the implementation of this scenario we used 10 students and 4 of each of the following: washing machines, dryers and keys. The only restraints are that the amount of resting people is not allowed to exceed a specific threshold. This threshold is calculated by taking the smaller number between the washing machines and dryers and subtracting one. For example, in this scenario the resting students can be a maximum of 3. The other restraint is that the barrier requires all of the students to enter before it is released. In this case the barrier waits until the `wait()` method is called 10 times before it opens the turnstile.

Implementation and Problems

Initial Implementation Attempts

The early versions of the laundromat simulation modeled students as independent threads that tried to acquire machines in sequence.
Students first obtained a **key** to enter the room, then reserved a **washing machine**, followed by a **dryer**.
During either washing or drying, students had a **70%** chance to rest - leaving the room temporarily and later reentering.

Initially, we protected the critical variables tracking "resting students" with simple counters and basic mutex locks.

Deadlock #1 - Stale Read Issue

We first encountered a stale read deadlock because of a wrapper class that we created. The idea was to wrap the counter for **resting students** in a class whose methods are protected by a mutex. The reasoning was that it made the code more readable and it would prevent dangling mutexes. Here is what went wrong:

- Students would read the count of **resting students** while it was still valid.
- Meanwhile, another thread would increment the counter between the read and the check.
- The first student would act based on an outdated assumption, leading to overbooking and system freeze.

Example:

```
1. class Wrapper
2.  def read(self):
3.      val = 0
4.      self.mutex.wait()
5.      val = self.val
6.      self.mutex.signal()
7.      return val
8.
9.  def increment(self)
10.     self.mutex.wait()
11.     self.val += 1
12.     self.mutex.signal()
13.     ...
14.
15. def student():
16. ...
17.     if wrapper.read() < n - 1:
18.         wrapper.increment()
19. ...
```

`N = 4`, start with `resting_students = 3`

Line	Thread	Description
17	A	Read *resting_students* = 3
17	B	Read *resting_students* = 3
18	B	Increment resting people to 4
18	A	Proceed as if it's 3, increment to 4, thus causing overwrite of the counter and overflow of students, which leads to a dealock

Fix: We quickly realized why this is not a standard practice and reversed to the previous version where the mutex simultaneously protects the read and write via inline operations and no wrapper class.

Deadlock #2 - Too Many Resting Students

Later, we tried limiting students resting after washing and after drying separately.
We allowed up to *(n−1)* students to pause per machine, meaning a total of 6 students could be out resting with appliances reserved.

Problem:

- More (or an equal amount) students are resting than there are keys.
- New students would block at `dryer.wait()` or `washer.wait()`, unable to progress.
- Deadlock occurred because no resting students were returning quickly enough to free appliances.

```
  # Student loop
1 keys.wait()
2 washing_machine.wait()

3 if decide_to_exit(0.7):
4   mutex.wait()
5   if people_resting_while_drying < n - 1:
6     people_resting_while_drying += 1
7     mutex.signal()
8     exit()
9     mutex.wait()
10    people_resting_while_drying -= 1
11 mutex.signal()

12 wash()
13 washing_machine.signal()
14 dryer.wait()

15 if decide_to_exit(0.7):
16   mutex.wait()
17   if people_resting_while_drying < n - 1:
18     people_resting_while_drying += 1
19     mutex.signal()
20     exit()
21     mutex.wait()
22     people_resting_while_drying -= 1
23   mutex.signal()

24 dry()
25 dryer.signal()
26 keys.signal()

# Exit function

27 def exit():
28   keys.signal()
29   keys.wait()
```

Fix: We globally limited the number of resting students across both washers and dryers to *(n−1)* total - not separately.

Fairness

To avoid starvation, we implemented a *Barrier* mechanism that coordinates student access to the laundromat. Before trying to take a key and enter the laundromat, each student must first pass through a waiting phase controlled by the barrier.
The barrier collects all students into a "waiting room" and releases them together, ensuring that no single student or group of students can monopolize access. No student can go on a second trip until every other has passed.

The *Barrier* works as follows:

```
class Barrier:
    def __init__(self):
        self.mutex = MySemaphore(1, "barrier mutex")
        self.count = 0
        self.turnstile = MySemaphore(0, "turnstile")

    def wait(self):
        self.mutex.wait()
        self.count += 1
        if self.count == STUDENTS:
            self.turnstile.signal(STUDENTS)
            self.count = 0
        self.mutex.signal()
        self.turnstile.wait()

...
while True:
    print(name + " is waiting")
    barrier.wait()
    keys.wait()
    print(name + " has entered the laundromat and is waiting for a washing machine")
...
```

How It Works:

- Each student first calls barrier.wait().
- The barrier tracks how many students have arrived by incrementing count.
- Once `count == STUDENTS`, the barrier opens the turnstile, allowing all students to proceed simultaneously.
- After passing through the turnstile, students attempt to acquire a key to enter the laundromat.

How It Ensures No Starvation

- Since all students must synchronize at the barrier before proceeding, no student can rush ahead unfairly.
- Fast or aggressive threads cannot starve slower ones because all must synchronize before the next cycle starts.
- Over time, this ensures bounded waiting time for all students.

Attempted Appliance Breaking Mechanism

We tried to implement a system where occasionally appliances break. The idea was that there are counters for the appliances and if one breaks it cannot be used. The counters keep track of how many washing machines/dryers can be used and when the counter is decreased, the semaphore is also decreased (by calling the `.wait()` method). To prevent deadlocks we allow less people to use the appliances by the following logic:

- For every washing machine that breaks we decrease the capacity (max_people_resting)
- For every dryer that breaks we decrease the capacity

- If a pair of washing machine and dryer break, we don't have to decrease by two people
  We implemented this by setting the `max_people_resting` to the minimum of the two counters decreased by one, as previously we chose to set the maximum allowed resting people to be `N - 1`. Furthermore we lowered the amount of people that can enter the system by decreasing the `keys` mutex as well. The values were safeguarded by the same mutex protecting `resting_people`.

Here is the code that we experimented with:

```
...
washing_machine.signal()
if chance(0.05):
    break_appliance("washing_machine")
...
```

```
1. def break_appliance(appliance):
2.     global keys, active_dryers, active_washing_machines, max_people_resting
3.     if appliance == "washing_machine":
4.         mutex.wait()
5.         active_washing_machines -= 1
6.         max_people_resting = min(active_washing_machines, active_dryers) - 1
7.         mutex.signal()
8.         washing_machine.wait()
9.     elif appliance == "dryer":
10.        mutex.wait()
11.        active_dryers -= 1
12.        max_people_resting = min(active_washing_machines, active_dryers) - 1
13.        mutex.signal()
14.        dryer.wait()
15.    keys.wait()

16. def repair_appliance(appliance):
17.     global keys, active_dryers, active_washing_machines, max_people_resting
18.     if appliance == "washing_machine":
19.         mutex.wait()
20.         active_washing_machines += 1
21.         max_people_resting = min(active_washing_machines, active_dryers) - 1
22.         mutex.signal()
23.         washing_machine.signal()
24.     elif appliance == "dryer":
25.         mutex.wait()
26.         active_dryers += 1
27.         max_people_resting = min(active_washing_machines, active_dryers) - 1
28.         mutex.signal()
29.         dryer.signal()
30.     keys.signal()
```

What went wrong

The breaking mechanism was triggered by a student thread. This means that the thread running the `break_appliance` function holds a key. If that thread waits at the `washing_machine` or `dryer` semaphore it then blocks an additional key. To solve this we would have had to introduce counters for the washing machines and keys and to consider edge cases like too many students resting for us to decrease the maximum allowed number of resting students.

Problem:

- A student thread that broke an appliance was still holding a key and blocked further operations.
- Reducing keys dynamically caused complicated dependencies between students, machines, and entry limits.
- Complexity became too high to manage safely within the existing system structure.

Decision: We removed appliance breaking from the final implementation to keep synchronization sound and verifiable.

Final Implementation

In the final version, the system enforces the following:

- **Keys Semaphore**: Students must acquire a key to enter the laundromat.
- **Strict Resource Order**: Washing machine must be acquired before dryer.
- **Controlled Resting**:
  - 70% chance to rest after starting washing or drying.
  - Resting students are globally limited to *(n−1)* regardless of which appliance they are using.
- **Mutex**: Used to protect reading and updating resting counters.
- **Exit and Reentry**: Students release their key when resting and must reacquire it before continuing.

With these design choices, the system achieves liveness (no deadlocks), fairness, and high throughput while allowing dynamic student behavior.

Code

```
from Environment import *
from Environment import _blk
import random, Environment as env, importlib

# ---------- Constants ----------

N = 4
STUDENTS = 10
CHANCE_OF_RESTING = 0.7

# ---------- Counters ----------

washing_machine_count = N
dryer_count = N
people_resting = 0

# ---------- Semaphores & Mutexes ----------
class Barrier:
    def __init__(self):
    self.mutex = MySemaphore(1, "barrier mutex")
    self.count = 0
    self.turnstile = MySemaphore(0, "turnstile")

def wait(self):
    self.mutex.wait()
    self.count += 1
    if self.count == STUDENTS:
        self.turnstile.signal(STUDENTS)
```

```
            self.count = 0
        self.mutex.signal()
        self.turnstile.wait()

barrier = Barrier()
mutex = MySemaphore(1, "mutex")
keys = MySemaphore(N, "keys")
washing_machine = MySemaphore(washing_machine_count, "washing_machines")
dryer = MySemaphore(dryer_count, "dryers")

# ---------- Utils ----------

def decide_to_exit(probability=0.5):
    return random.random() < probability

def try_to_exit(name: str):
    global people_resting
    print(name + " wants to rest")

    mutex.wait()
    if people_resting < N - 1:
        people_resting += 1
        mutex.signal()

        print(name + " exited the laundromat")
        exit()
        print(name + " returned to the laundromat")

        mutex.wait()
        people_resting -= 1
    mutex.signal()

def exit():
    keys.signal()
    print("Exiting laundromat")
    keys.wait()

# ---------- Threads ----------

student_n = 0
def student_thread():
    mutex.wait()
    student_n += 1
    mutex.release()

    name = "Student " + str(student_n)
    print(name + " has started")

    while True:
        print(name + " is waiting")
        barrier.wait()
        keys.wait()
        print(name + " has entered the laundromat and is waiting for a washing machine")

        washing_machine.wait()
        print(name + " has started a washing machine")
```

```
        if decide_to_exit(CHANCE_OF_RESTING):
            try_to_exit(name)

        washing_machine.signal()
        print(name + " has finished a washing machine and is waiting for a dryer")
        dryer.wait()

        if decide_to_exit(CHANCE_OF_RESTING):
            try_to_exit(name)

        dryer.signal()
        keys.signal()
        print(name + " has finished a dryer and left the laundromat")

# ---------- Set up ----------

def createThreads(numberOfThreads, threadFunction):
    for _ in range(numberOfThreads):
        subscribe_thread(threadFunction)

def setup():
    createThreads(10, student_thread)

if __name__ == '__main__':
    importlib.import_module("Laundromat").setup()

    env.GuiCreate("Laundromat.py")
    env.GuiMainloop()
```

Conclusion

Visualizing

To visualize the behavior of our concurrent laundromat system, we selected a UML sequence diagram as the most appropriate modeling tool. This choice was based on its ability to explicitly represent interactions over time, making it ideal for tracking threads (students), shared resources (keys, washers, dryers), and synchronization points (barriers and semaphores).

Sequence diagrams are particularly suited to concurrency modeling because they make the temporal order of operations clear-especially when multiple lifelines interact asynchronously with shared objects. As discussed in the *OMG UML 2.6 Specification (2012)*, sequence diagrams allow for the depiction of concurrent behaviors using vertical message flows and combined fragments such as par and alt, enabling accurate modeling of nondeterminism, blocking, and race conditions.

StackOverflow contributors also highlight sequence diagrams as one of the few UML types explicitly equipped to handle thread interactions and synchronization in concurrent systems (StackOverflow, 2024). In our implementation, the diagram captures key decision points-such as resting behavior and entry/exit synchronization-while maintaining clarity in the face of looping and conditional behavior.

This project explored the synchronization challenges of managing shared resources in a laundromat scenario, modeled closely after classic concurrency problems like the Dining Philosophers.
Starting from an intuitive but naive implementation, we encountered and resolved multiple deadlock scenarios caused by stale reads, incorrect resting limits, and poorly coordinated resource handling.

Through careful iteration, we established a robust system based on:

- Strict ordering of machine usage (washer before dryer),
- A key-based access system limiting the number of active students,
- Global control of resting students with mutex-protected counters,
- Safe entry and exit operations for students temporarily leaving the room.
- Prevention of starvation regardless of the type of scheduler.

Ultimately, the final design successfully prevents deadlocks, ensures fairness among students, and maintains high system throughput under dynamic and probabilistic behavior.
The principles and solutions developed here can serve as a reference for designing synchronization mechanisms in broader distributed or resource-constrained systems.

Research and Related Work

Dining Philosophers and other problems

The laundromat synchronization problem shares fundamental characteristics with classical concurrency problems, most notably the *Dining Philosophers Problem*, which is often cited as a model for understanding deadlock, mutual exclusion, and fairness in shared-resource systems. One key difference between this and our problem is that in the *Dining Philosophers* the resources can be obtained in a different order (fork 2 may be obtained after fork 1 or before fork 3), thus opening the possibility for a deadlock where two philosophers obtain all available forks beside the one between them. In the laundromat resources (washing machine and dryer) are sequential which means that a student may not acquire a dryer before a washing machine, so this type of deadlock is not possible.

Additionally, our implementation of fairness draws from well-established patterns described in Downey's *The Little Book of Semaphores* (2016). In particular, the **Barrier** pattern (p. 21) was adapted to enforce collective entry into the laundromat, ensuring that all students advance through the simulation in a fair and synchronized manner. This mechanism prevents starvation by batching access attempts, which is a direct response to issues that arise in non-protected semaphore usage.

Downey also discusses problems such as *Dining Philosophers* (p. 87) and *Dining Savages* (p. 115), both of which helped shape our approach to deadlock prevention and shared resource coordination. By referencing these classical problems, we ground our simulation in established concurrency theory while extending it to a novel and practical scenario.

References

Downey, A. (2016). *The Little Book of Semaphores* (2nd ed.). Retrieved from https://github.com/AllenDowney/LittleBookOfSemaphores

- **Barrier** data structure: page *21*
- **Dining Philosophers** problem: page *87*
- **Dining Savages** problem: page *115*

Franco, F. (2020, May 27). *Dining Philosophers Problem*. Medium. https://medium.com/@francescofranco_39234/dining-philosophers-problem-36d0030a4459

Object Management Group. (2012). *Concurrency in UML Version 2.6*. https://www.omg.org/certification/uml/documents/concurrency_in_uml_version_2.6.pdf

StackOverflow. (2024). *What UML diagrams are available for modelling concurrent programs and systems?* https://stackoverflow.com/questions/76282735/what-uml-diagrams-are-available-for-modelling-concurrent-programs-and-systems