Operating Systems and Networks SoSe 25 Notes

Igor Dimitrov

2024-12-18

Table of contents

Preface

1 Process Management

1.1 Condition Variables and Producer / Consumer Problem

Condition variables are employed **together** with mutexes when synchronizing producers and consumers. It woul be incorrect ot only use a condition variable without a mutex, or a mutex with busy waiting without a condition varible.

Incorrect Variant 1: Condition Variable Without Mutex

```
ready = False
condition = ConditionVariable()

def wait_thread():
    if not ready:
        condition.wait()  # Incorrect: no mutex guarding shared state
    print("Condition met!")

def signal_thread():
    ready = True
    condition.notify()
```

Why It's Wrong:

- Access to `ready` is unprotected — race conditions may occur.
- `condition.wait()` must always be used with a mutex.

Incorrect Variant 2: Mutex Without Condition Variable (Busy Waiting)

```
ready = False
mutex = Mutex()
```

```
def wait_thread():
    while True:
        mutex.lock()
        if ready:
            mutex.unlock()
            break
        mutex.unlock()
        sleep(0.01)  # Active polling (wasteful)

def signal_thread():
    mutex.lock()
    ready = True
    mutex.unlock()
```

Why It's Problematic:

- Avoids races, but wastes CPU via busy waiting.
- Also prone to subtle visibility issues if memory barriers aren't enforced.

Correct Variant: Condition Variable with Mutex

```
ready = False
mutex = Mutex()
condition = ConditionVariable()

def wait_thread():
    mutex.lock()
    while not ready:
        condition.wait(mutex)  # Atomically unlocks and waits
    mutex.unlock()
    print("Condition met!")

def signal_thread():
    mutex.lock()
    ready = True
    condition.notify()
    mutex.unlock()
```

Why It Works:

- Shared state is properly guarded.

- No busy waiting.
- Safe signaling and waking.

Another question is why to use `while not ready` and not simply `if not ready`:

```
def wait_thread():
    mutex.lock()
    if not ready:
        condition.wait(mutex)
    mutex.unlock()
```

Problem:

- May miss spurious wakeups or situations where multiple threads wait and only one should proceed.
- A `while` loop is necessary to recheck the condition after being woken up.

Producer/Consumer Problem

Variant A: Unbounded Queue (No Buffer Limit)

```
queue = []
mutex = Mutex()
not_empty = ConditionVariable()

def producer():
    while True:
        item = produce()
        mutex.lock()
        queue.append(item)
        not_empty.notify()
        mutex.unlock()

def consumer():
    while True:
        mutex.lock()
        while not queue:
            not_empty.wait(mutex)
        item = queue.pop(0)
```

```
        mutex.unlock()
        consume(item)
```

Variant B: Bounded Queue (Fixed Buffer Size)

```
queue = []
BUFFER_SIZE = 10
mutex = Mutex()
not_empty = ConditionVariable()
not_full = ConditionVariable()

def producer():
    while True:
        item = produce()
        mutex.lock()
        while len(queue) >= BUFFER_SIZE:
            not_full.wait(mutex)
        queue.append(item)
        not_empty.notify()
        mutex.unlock()

def consumer():
    while True:
        mutex.lock()
        while not queue:
            not_empty.wait(mutex)
        item = queue.pop(0)
        not_full.notify()
        mutex.unlock()
        consume(item)
```

1.2 Summary Table

Case	Uses Mutex	Uses Condition Variable	Blocking	CPU-Efficient	Correct
1. Condition variable without mutex	No	Yes	No	Yes	No
2. Mutex without condition variable	Yes	No	No	No (busy)	Partly
3. Condition variable with mutex	Yes	Yes	Yes	Yes	Yes
4. If instead of while	Yes	Yes	Yes	Yes	Risky
5. Producer/Consumer (unbounded)	Yes	Yes (`not_empty`)	Yes	Yes	Yes
6. Producer/Consumer (bounded)	Yes	Yes (`not_empty`, `not_full`)	Yes	Yes	Yes

---

Operations of a Bounded Queue

Step	Operation	`in`	`out`	Buffer State	Count == (`(in - out + 5) % 5`)
0	Start	0	0	`[_ _ _ _ _]`	0
1	Produce A	1	0	`[A _ _ _ _]`	1
2	Produce B	2	0	`[A B _ _ _]`	2
3	Produce C	3	0	`[A B C _ _]`	3
4	Consume → A	3	1	`[_ B C _ _]`	2
5	Consume → B	3	2	`[_ _ C _ _]`	1
6	Produce D	4	2	`[_ _ C D _]`	2
7	Produce E	0	2	`[_ _ C D E]`	3
8	Consume → C	0	3	`[_ _ _ D E]`	2
9	Produce F	1	3	`[F _ _ D E]`	3

where

- `in`: the write position / index
- `out`: the read position /index
- `count == (in - out + 5) % 5` is the invariant of the data structure, giving the number of elements in the buffer