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

2 Memory Management

2.1 Virtual Memory

Paging: Translating Logical to Physical Addresses

Context

In paging, the operating system divides:

- Logical (virtual) memory into fixed-size pages
- Physical memory (RAM) into same-size frames

Each process has a page table that maps page numbers to frame numbers.

Our goal is:

> Given a virtual address, compute the corresponding physical address.

Example Setup

- Virtual address $V = 7000$
- Page size = 4096 bytes = 2^{12} $k = 12$
- Assume the page table maps page 1 to frame 9: $F(1) = 9$

Step 1: Manual (Arithmetic) Calculation

To translate a virtual address manually, we need to answer two questions:

1. **Which page** is the address in?
2. **Where within that page** is the address?

This is done by:

- Dividing the address by the page size to get the **page number**
- Taking the remainder (modulo) to get the **offset** within the page

Apply this to $V = 7000$ with page size 4096:

- Page number $p = \left\lfloor \frac{7000}{4096} \right\rfloor = 1$
- Offset $d = 7000 \mod 4096 = 2904$

Now we look up page 1 in the page table:

- Frame number $f = F(1) = 9$

To get the final physical address, we compute the base address of frame 9 and add the offset:

- Physical address $= f \cdot 4096 + d = 9 \cdot 4096 + 2904 = 39768$

Result: 39768

Step 2: Bitwise Calculation (Optimized for Hardware)

For power-of-two page sizes, the address can be efficiently split using bitwise operations:

- Page number $= V \gg 12$ (right shift by 12 bits is equivalent to dividing by 4096)
- Offset $= V \& (2^{12} - 1) = V \& 0xFFF$ (bit mask keeps the lower 12 bits)
- Page number 1 maps to frame number $f = F(1) = 9$

To compute the **frame's starting address**, we use a left shift:

- $f \ll 12 = 9 \ll 12 = 36864$, which is equivalent to $9 \cdot 4096$

Final physical address:

- Physical Address $= 36864 + 2904 = 39768$

Same result, now using fast bit operations.

Bit Sequence Visualization

Let's visualize how the virtual address is split in binary:

- Virtual address $V = 7000$
- Binary representation (14 bits): `0001 1011 0101 1000`

Split into:

- Page number (upper 2 bits): `00 01` $\rightarrow 1$
- Offset (lower 12 bits): `1011 0101 1000` $\rightarrow 2904$

This split works because:

- The lower 12 bits represent the offset for a 4 KB page
- The upper bits index into the page table

Why This Works Mathematically

The logic behind using bit shifts and masks instead of division and modulo is based on how numbers are represented in binary.

Decimal Analogy (Base 10)

Consider dividing `1375` by powers of 10:

- $137\mathbf{5} \div 10^1 = 137 \quad (\text{modulo: } \mathbf{5})$
- $13\mathbf{75} \div 10^2 = 13 \quad (\text{modulo: } \mathbf{75})$
- $1\mathbf{375} \div 10^3 = 1 \quad (\text{modulo: } \mathbf{375})$

The rightmost digits are the remainder (modulo); the left are the quotient (division).

Binary Example (Base 2)

Take the binary number `1011` ($= 11$):

- $101\mathbf{1} \div 2^1 = 101 = 5 \quad (\text{modulo: } \mathbf{1})$
- $10\mathbf{11} \div 2^2 = 10 = 2 \quad (\text{modulo: } \mathbf{11} = 3)$
- $\mathbf{1011} \div 2^3 = 1 = 1 \quad (\text{modulo: } \mathbf{011} = 3)$

In both systems, the rightmost digits/bits represent the **offset**, and the leftmost represent the **page number**.

This is why in binary:

- $V \gg k$ is equivalent to $\lfloor V/2^k \rfloor$
- $V\&(2^k - 1)$ is equivalent to $V \mod 2^k$
- $f \ll k$ is equivalent to $f \cdot 2^k$, which gives the frame base address

These operations are both mathematically correct and hardware-efficient.

Final Formula

$$\text{Physical Address} = (F(V \gg k) \ll k) + (V \& (2^k - 1))$$

This computes:

- The page number via right shift
- The frame number from the page table
- The frame base via left shift (i.e., multiplying by page size)
- The final physical address by adding the offset

Additional Example for Practice and Clarity

Let's now take another address and apply all three methods for reinforcement.

Setup

- Virtual address $V = 13{,}452$
- Page size $= 4096 = 2^{12}$
- Page table:

Page #	Frame #
0	3
1	7
2	1
3	6

Manual Calculation

- Page number: $13{,}452 \div 4096 = 3$
- Offset: $13{,}452 \mod 4096 = 1164$
- Frame number: $F(3) = 6$
- Physical address $= 6 \cdot 4096 + 1164 = 24{,}576 + 1164 = 25{,}740$

Bitwise Calculation

- $V = 13{,}452 = 0b0011\ 0100\ 1001\ 1100$
- Page number $= V \gg 12 = 3$
- Offset $= V \& 0xFFF = 1164$
- Frame number $= F(3) = 6$
- Frame base $= 6 \ll 12 = 24{,}576$
- Physical address $= 24{,}576 + 1164 = 25{,}740$

Using the Formula

$$\text{Physical Address} = (F(V \gg 12) \ll 12) + (V \& 0xFFF)$$

$$= (6 \ll 12) + 1164 = 24{,}576 + 1164 = 25{,}740$$

Conclusion

When the page size is a power of two, address translation can be performed using fast bit operations instead of division and modulo. This is possible because of how binary numbers encode positional value. We saw that the lower bits give the offset and the upper bits the page number. Whether done manually, with bit operations, or using the translation formula, all approaches yield the same physical address — and this consistency is what makes paging both robust and efficient.