

Operating Systems and Networks SoSe 25

Notes

Igor Dimitrov

2024-12-18

Table of contents

Preface	3
1 Process Management	4
1.1 Condition Variables and Producer / Consumer Problem	4
Incorrect Variant 1: Condition Variable Without Mutex	4
Incorrect Variant 2: Mutex Without Condition Variable (Busy Waiting)	4
Correct Variant: Condition Variable with Mutex	5
Producer/Consumer Problem	6
Variant B: Bounded Queue (Fixed Buffer Size)	7
1.2 Summary Table	7
2 Memory Management	9
2.1 Virtual Memory	9
Paging: Translating Logical to Physical Addresses	9
Single-Level (Direct) Page Tables	13
Why Single-Level Tables Are Impractical	14
Frame Table (Global Physical Memory Tracking)	14
Inverted Page Tables	15
Hierarchical Page Tables	15
When and How Page Tables Are Allocated	16
Physical Memory: Frames and Their Usage	17
Kernel Mapping and Access	17
Translation Lookaside Buffer	18
Final Thoughts	18

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 would be incorrect to only use a condition variable without a mutex, or a mutex with busy waiting without a condition variable.

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	CPU- BlockingEfficient	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 (<code>not_empty</code>)	Yes	Yes	Yes
6. Producer/Consumer (bounded)	Yes	Yes (<code>not_empty</code> , <code>not_full</code>)	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 $p = \lfloor \frac{7000}{4096} \rfloor = 1$
- Offset $d = 7000 \bmod 4096 = 2904$

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:

- $1375 \div 10^1 = 137$ (modulo: **5**)
- $1375 \div 10^2 = 13$ (modulo: **75**)
- $1375 \div 10^3 = 1$ (modulo: **375**)

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

Binary Example (Base 2)

Take the binary number 1011 (= 11):

- $1011 \div 2^1 = 101 = 5$ (modulo: **1**)
- $1011 \div 2^2 = 10 = 2$ (modulo: **11** = 3)
- $1011 \div 2^3 = 1 = 1$ (modulo: **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 $V \bmod 2^k$
- $f \ll k$ 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: $13,452 \bmod 4096 = 1164$
- 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.

Single-Level (Direct) Page Tables

In the simplest form of paging, each process has its own **single-level page table**, which directly maps **virtual page numbers** to **physical frame numbers**.

For example, in a system with:

- A **32-bit virtual address space** (4 GB total)
- A **page size of 4 KB = 2^{12} bytes**

The number of virtual pages is:

$$2^{32}/2^{12} = 2^{20} = 1,048,576 \text{ entries}$$

If each page table entry (PTE) is 4 bytes, the total size of the page table is:

$$2^{20} \times 4 = 4 \text{ MB per process}$$

In a **64-bit system**, even with larger pages (e.g. 4 MB), the number of virtual pages is so large (e.g., 2^{52}) that flat page tables become **completely impractical**.

Why Single-Level Tables Are Impractical

Main issues:

- **Memory usage per process** becomes excessive (e.g., 4 MB/page table \times hundreds of processes)
- **Scaling issues** as address spaces grow
- Most processes **use only a small part** of their virtual address space, so allocating full page tables is wasteful

Thus, alternative paging strategies are needed.

Frame Table (Global Physical Memory Tracking)

The OS maintains a **frame table**, which tracks:

- Which physical frames are in use
- What each frame is used for (user page, kernel structure, page table, etc.)
- Associated metadata: dirty bit, reference count, owner process

This allows the OS to **allocate and deallocate physical memory** intelligently and safely, and is crucial for page replacement, memory protection, and I/O operations.

Inverted Page Tables

Instead of one page table per process, an **inverted page table** contains **one entry per physical frame**. Each entry stores:

- **Process ID**
- **Virtual page number** mapped to this frame

The index in the table corresponds to the **physical frame number**.

Benefits:

- Memory usage depends only on **RAM size**, not virtual address space

Drawbacks:

- To resolve a virtual address, the system must **search (or hash)** the inverted table to find the matching (PID, VPN) pair
- More complex lookup; slower than indexed access
- Difficult to implement efficient TLB caching

Used in some older or memory-constrained systems, but rare in modern general-purpose OSes.

Hierarchical Page Tables

Modern systems (e.g., x86, Linux, Windows) use **multi-level page tables** to address sparsity in virtual memory usage.

The virtual address is divided into parts:

- Each part indexes a **level in the page table hierarchy**
 - Only needed page table levels are **allocated on demand**
-

32-bit Two-Level Paging Example

For a 32-bit address space with 4 KB pages:

$$\text{Virtual address} = [10 - \text{bitPageDir}][10 - \text{bitPageTable}][12 - \text{bitOffset}]$$

- 1024 entries in the **Page Directory (PD)**
- Each PD entry points to a **Page Table (PT)** with 1024 entries
- Final PT entry points to the **physical frame**

Only PD is allocated initially. PTs are created **only when needed**, saving memory.

64-bit Four-Level Paging Example (x86-64)

Typically uses **48-bit virtual addresses**, split into:

$$[9 - \text{bitPML4}][9 - \text{bitPDPT}][9 - \text{bitPD}][9 - \text{bitPT}][12 - \text{bitoffset}]$$

Each level (PML4, PDPT, PD, PT) is 512 entries (9 bits).

This allows mapping of:

$$512^4 \times 4 \text{ KB} = 256 \text{ TB of virtual address space}$$

Page tables are allocated **only as needed** — i.e., when a virtual page is first accessed.

When and How Page Tables Are Allocated

Page tables are allocated in two situations:

1. **At program load time**

- When the OS loads a process, it allocates top-level tables and reserves virtual regions for code, data, stack, etc.
- Not all intermediate tables are allocated yet.

2. **On demand via page faults**

- When a process accesses a virtual address with no current mapping, the CPU triggers a **page fault**
 - The OS checks if the access is valid (e.g., within heap or stack)
 - If valid:
 - Missing intermediate page tables are allocated
 - A physical frame is allocated
 - Page tables are updated
 - Execution resumes
-

Physical Memory: Frames and Their Usage

RAM is divided into fixed-size **frames** (e.g., 4 KB). Each frame can hold:

- A user page (code, stack, heap)
- A page table (of any level)
- A kernel structure
- Anything the OS needs

Important: **contiguous physical frames may contain totally unrelated contents**, from different processes or different kinds of structures. The OS manages this via the frame table.

Kernel Mapping and Access

- Kernel code and data are mapped into the **upper portion of each process's virtual address space**
 - E.g., in 32-bit Linux: 0xC0000000 - 0xFFFFFFFF
- This region is **not accessible in user mode** (protected by page table flags)

Benefits:

- Enables **fast system calls and interrupt handling** without switching page tables
- Avoids need to reload CR3 (page table base register) on each syscall

Kernel runs entirely in **kernel mode**, and its physical location is:

- Not fixed
- Determined at boot (by the bootloader)
- Mapped into kernel virtual memory

Modern systems may use **KASLR (Kernel Address Space Layout Randomization)** to randomize the kernel's location for security.

Translation Lookaside Buffer

The TLB is a small, fast cache used by the CPU's MMU to store **recent virtual-to-physical translations**.

Why it's needed:

- Page table lookups involve multiple memory accesses (especially in multi-level schemes)
- TLB provides **near-instant translation** for repeated accesses

Effectiveness:

- Very high hit rate due to **spatial and temporal locality**
 - TLBs are usually 16–512 entries
 - Managed by the hardware; updated on TLB misses
-

Final Thoughts

Modern virtual memory management systems are built around the idea of:

- **Sparse usage** of large address spaces
- **On-demand, hierarchical translation structures**
- **Fast caching and protection** mechanisms

Direct page tables are conceptually simple but infeasible for large systems. Hierarchical page tables, supported by frame tables and TLBs, provide the scalability, flexibility, and efficiency required in today's general-purpose operating systems.
