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 = $2^{12} \quad k = 12$
- 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 $d = 7000 \mod 4096 = 2904$

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` $(= 11\ \ )$:

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

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

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

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.

Absolutely — here is the **complete regenerated summary**, now incorporating:

- The **updated "Inverted Page Tables"** section with size explanation and example
- The **updated "Hierarchical Page Tables"** section with both the 32-bit and 64-bit address resolution examples and definitions of each table level
- Consistent formatting throughout:
  - **No bold in headers**
  - **Minimal boldface emphasis** in the text — used only where strictly useful for clarity

This version is fully ready for integration into your Quarto notes.

Page Tables

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 global 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. It is also crucial for page replacement algorithms, memory protection, and managing shared pages or I/O buffers.

Inverted Page Tables

In a traditional page table system, each process maintains its own page table, which maps virtual pages to physical frames. In contrast, an inverted page table uses a fundamentally different approach:

- There is a single global page table for the entire system
- It contains one entry for each physical frame, not for each virtual page
- Each entry records:
  - The process ID that owns the frame
  - The virtual page number that maps to it
  - Any additional metadata (e.g., access flags, validity)

This approach dramatically reduces memory overhead, especially in systems with large virtual address spaces.

Size of the inverted page table

The size of an inverted page table depends only on the number of physical frames, which is determined by:

$$\text{Number of entries} = \frac{\text{RAM size}}{\text{frame size}}$$

Each entry stores fixed-size metadata (such as PID and VPN), so the total size is:

$$\text{Table size} = \frac{\text{RAM size}}{\text{frame size}} \times \text{entry size}$$

The ratio of the table size to RAM size simplifies to:

$$\frac{\text{Table size}}{\text{RAM size}} = \frac{\text{entry size}}{\text{frame size}}$$

This ratio is independent of total RAM size. In other words, the memory overhead of the page table scales proportionally with RAM but is bounded by the frame size and the entry size.

Concrete example

Consider a 32-bit system with the following properties:

- Physical RAM: 4 GB = 2^{32} bytes
- Page/frame size: 4 KB = 2^{12} bytes
- Page table entry size: 8 bytes (to store PID, VPN, flags, etc.)

Step-by-step:

1. Number of physical frames:

$$\frac{2^{32}}{2^{12}} = 2^{20} = 1{,}048{,}576 \text{ frames}$$

2. Total inverted page table size:

$$2^{20} \times 8 = 8 \text{ MB}$$

3. Relative overhead:

$$\frac{8 \text{ MB}}{4 \text{ GB}} = \frac{1}{512}$$

This means the page table occupies only about 0.2% of RAM.

Summary of trade-offs

Inverted page tables offer substantial memory savings, especially on systems with large or sparsely used virtual address spaces. However, the downside is that address translation becomes more complex:

- The system must search (or hash) the page table to find the matching (process ID, virtual page) pair
- This lookup is slower than direct indexing
- TLB caching becomes less straightforward

For this reason, inverted page tables are rarely used in modern general-purpose OSes, though they are still valuable in embedded or resource-constrained systems.

Hierarchical Page Tables

Modern systems (e.g., x86, Linux, Windows) use multi-level page tables to avoid allocating massive single-level tables for sparse address spaces. The key idea is to divide the virtual address into multiple segments, each of which indexes a level in the page table hierarchy. This allows the OS to only allocate memory for regions that are actually used.

Each level of the hierarchy resolves part of the virtual address and points to the next level down. The final level contains the physical frame number. The remaining bits (the offset) are added to form the final physical address.

This approach reduces memory overhead and supports sparse, large virtual address spaces.

32-bit Two-Level Paging Example

Assume a 32-bit virtual address space with:

- Page size = 4 KB = 2^{12}
- 10 bits for the page directory index
- 10 bits for the page table index
- 12 bits for the offset

The virtual address layout is:

```
[ 10 bits | 10 bits | 12 bits ]
   PD index  PT index   Offset
```

Suppose the virtual address is:

```
VA = 0x1234ABCD
```

Convert to binary:

```
0001 0010 0011 0100 1010 1011 1100 1101
```

Split:

- Page Directory index = `0001001000` = 0x048 = 72
- Page Table index = `1101001010` = 0x34A = 842
- Offset = `101111001101` = 0xBCD = 3021

Assume:

- The page directory is located at physical address `0x00100000`
- Entry 72 in the page directory points to a page table at `0x00200000`
- Entry 842 in that page table points to a physical frame at `0x00ABC000`

Final physical address:

$$0x00ABC000 + 0xBCD = 0x00ABCBCD$$

This example illustrates how a 2-level table hierarchy resolves the virtual address to a physical address through two indirections and an offset.

64-bit Four-Level Paging Example

Modern x86-64 systems typically support a 48-bit virtual address space, split across four paging levels. The page size remains 4 KB = 2^{12}.

Each level of the hierarchy resolves 9 bits (since $2^9 = 512$ entries per table), so the full 48-bit address is broken into:

```
[ 9 bits | 9 bits | 9 bits | 9 bits | 12 bits ]
  PML4     PDPT     PD       PT       Offset
```

The levels are defined as follows:

- PML4 (Page Map Level 4): The root of the page table hierarchy; indexed by the top 9 bits of the virtual address. Each entry points to a PDPT.
- PDPT (Page Directory Pointer Table): Intermediate level; each entry points to a Page Directory.

- PD (Page Directory): Each entry points to a Page Table.
- PT (Page Table): Final level; each entry contains a physical frame number.
- Offset: Specifies the exact byte within the 4 KB page.

Suppose the virtual address is:

```
VA = 0x00007F34_1234ABCD
```

Breaking down the lower 48 bits:

- PML4 index = bits 47–39 = 0
- PDPT index = bits 38–30 = 505
- PD index = bits 29–21 = 322
- PT index = bits 20–12 = 210
- Offset = bits 11–0 = 0xAF3D = 44861

Assume the following physical mappings:

- CR3 register points to PML4 at `0x00100000`
- PML4 entry 0 → PDPT at `0x00200000`
- PDPT entry 505 → PD at `0x00300000`
- PD entry 322 → PT at `0x00400000`
- PT entry 210 → frame at `0x00ABC000`

Final physical address:

$$0x00ABC000 + 0xAF3D = 0x00B06F3D$$

This example demonstrates how a virtual address is translated step-by-step through four levels of indirection. The layered structure supports extremely large address spaces (up to 256 TB) without requiring full allocation of all intermediate tables.

Summary

Hierarchical page tables solve the scalability problem of flat page tables by breaking the virtual address into multiple segments. Each level of the hierarchy is a smaller table, and lower levels are allocated only when needed. This provides a sparse, memory-efficient structure for address translation.

The cost of additional indirection is mitigated by the use of TLBs, which cache recent address translations to avoid repeated page walks.

When and How Page Tables Are Allocated

Page tables are allocated in two situations:

1. At program load time The OS allocates top-level tables and reserves virtual address regions for code, data, stack, etc., but not necessarily all intermediate tables.
2. On demand via page faults When a process accesses a virtual address with no current mapping, the CPU triggers a page fault. If the access is valid, the OS allocates missing intermediate page tables and a physical frame, updates the page table entries, and resumes execution.

This approach enables sparse memory allocation and efficient use of physical memory.

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
- Other memory-resident objects

Contiguous physical frames may contain completely unrelated contents, as physical memory management is modular and page-based. The OS tracks frame usage via the global frame table.

Kernel Mapping and Access

The kernel is mapped into the upper region of each process's virtual address space (e.g., from `0xC0000000` upward in 32-bit systems). This allows:

- Fast system calls and interrupt handling
- Avoiding page table switches on mode transitions

This region is protected by page table flags, preventing access in user mode. The kernel itself runs entirely in kernel mode. Its physical location is determined at boot and may vary across systems. Techniques like KASLR (Kernel Address Space Layout Randomization) add further protection.

Translation Lookaside Buffer

The TLB is a hardware-managed cache used by the MMU to store recently used virtual-to-physical page translations.

Why it's important:

- Page walks involve multiple memory accesses
- The TLB allows near-instant translation on a hit
- Reduces the average cost of memory access in the presence of multi-level page tables

TLBs are small (typically 16–512 entries) but highly effective due to temporal and spatial locality in most program behavior.