華中師範大學
CENTRAL CHINA NORMAL UNIVERSITY

操作系统实验报告

第六次实验：SPOOLING 输出系统设计与实现

姓名 赖景康

学号 2023214564

课程 操作系统

学院 计算机学院

2025 年 12 月 28 日

1 实验目的与要求

- 理解并掌握 SPOOLING 技术（脱机输出的假脱机机制），区分用户进程、SPOOLING 输出进程与外设驱动的职责边界。
- 在并发环境下实现“多生产者 →SPOOLING→ 设备”的安全流水线，确保批次边界正确、顺序一致且设备高效吞吐。
- 以 Rust 实现线程与通道通信，验证批次输出、关机协议与资源回收的正确性。

2 实验环境

- 开发工具：VS Code
- 编程语言：Rust 1.70+（Edition 2021）
- 项目位置：`task6/`

3 原理与设计

3.1 SPOOLING 原理

SPOOLING（Simultaneous Peripheral Operation On-Line）通过在内存/磁盘上设置缓冲区（“输出井”），将用户进程对慢速外设的直接访问改为对缓冲的顺序提交：

1. **用户进程**产生输出请求，逐条写入到 SPOOLING 进程；
2. **SPOOLING 进程**按请求的批次边界装配为“输出块”，并顺序发送至设备；
3. **外设线程**（打印机/CRT）只需线性消费块，避免与用户进程直接并发争用。

该模式减少上下文切换与设备访问的随机性，提高吞吐并保证批次完整性。

3.2 架构与数据流

本实验采用三类线程与两级通道：

- **Producer**：多个并发用户进程，生成 `OutputRequest`（含 `pid`、`content`、`end_batch`）。
- **Spooling**：集中搬运与装配批次，遇到 `end_batch` 即打包为 `OutputBlock` 并下发。
- **Device**：顺序消费 `OutputBlock`，串行写出到外设（此处以终端打印模拟）。
- 通道：`mpsc::channel<SpoolMessage>`（Producer→Spooling）、`mpsc::channel<OutputBlock>`（Spooling→Device）。

3.3 关键协议

- **批次边界**：每个批次的最后一条请求设置 `end_batch=true`，SPOOLING 形成块并发送。
- **关机信号**：使用 `SpoolMessage::Shutdown` 通知 SPOOLING 完成收尾并关闭下游通道，促使设备线程优雅退出。
- **容错处理**：当设备通道关闭或发送失败，SPOOLING 记录错误并停止进一步输出以避免悬挂。

4 核心代码展示

为保持与代码一致性，以下直接引用项目源文件（Rust）：

文件：`task6/src/main.rs`

```rust
use rand::{distributions::Alphanumeric, Rng};
use std::sync::mpsc::{self, Receiver, Sender};
use std::thread;
use std::time::Duration;

#[derive(Clone, Debug)]
struct OutputRequest {
    pid: u32,
    content: String,
    end_batch: bool,
}

#[derive(Clone, Debug)]
struct OutputBlock {
    batch_id: u64,
    items: Vec<OutputRequest>,
}

#[derive(Debug)]
enum SpoolMessage {
    Request(OutputRequest),
    Shutdown,
}

/// 设备线程：消费输出块，模拟打印机/CRT 输出
```

```rust
fn device_thread(rx_dev: Receiver<OutputBlock>) {
    while let Ok(block) = rx_dev.recv() {
        println!("\n=== 输出设备: 接收批次 {}, 条目 {} ===", block.batch_id,
            block.items.len());
        for (i, item) in block.items.iter().enumerate() {
            println!("[Batch {} / Item {}] 来自进程 {}: {}", block.batch_id, i +
                1, item.pid, item.content);
        }
        println!("=== 批次 {} 输出完成 ===\n", block.batch_id);
    }
    println!("输出设备: 通道关闭, 设备退出。");
}

/// SPOOLING 输出进程: 每次搬运一条信息到输出井; 遇到结束标志时,
/// 将当前井中的内容形成一个输出块并发送到设备。
fn spooling_thread(rx_req: Receiver<SpoolMessage>, tx_dev: Sender<OutputBlock>)
    {
    let mut current_batch: Vec<OutputRequest> = Vec::new();
    let mut batch_counter: u64 = 0;

    while let Ok(msg) = rx_req.recv() {
        match msg {
            SpoolMessage::Shutdown => {
                // 若井中尚有未输出的数据, 可选择进行一次尾块输出 (可选)
                if !current_batch.is_empty() {
                    batch_counter += 1;
                    let block = OutputBlock {
                        batch_id: batch_counter,
                        items: std::mem::take(&mut current_batch),
                    };
                    let _ = tx_dev.send(block);
                }
                break;
            }
            SpoolMessage::Request(req) => {
                // 每次只搬运一条 (模拟“每运行一次输出一项信息到输出井”)
                println!("SPOOLING: 接收来自进程 {} 的一条信息, end_batch = {}",
                    req.pid, req.end_batch);
                current_batch.push(req.clone());
```

```
            if req.end_batch {
                batch_counter += 1;
                let block = OutputBlock {
                    batch_id: batch_counter,
                    items: std::mem::take(&mut current_batch),
                };
                println!("SPOOLING: 形成输出块（批次 {}，条目 {}），发送到设备
                    ", block.batch_id, block.items.len());
                if tx_dev.send(block).is_err() {
                    eprintln!("SPOOLING: 设备通道已关闭，无法输出。终止。");
                    break;
                }
            }
        }
    }
    // 关闭设备通道，提示设备退出
    drop(tx_dev);
    println!("SPOOLING: 退出。");
}

/// 产生输出请求的进程：随机生成若干条消息，每个批次以结束标志收尾
fn producer_thread(pid: u32, tx_req: Sender<SpoolMessage>, batches: usize) {
    let mut rng = rand::thread_rng();

    for b in 0..batches {
        // 每个批次随机生成 3..6 条信息
        let items_in_batch = rng.gen_range(3..=6);
        for i in 0..items_in_batch {
            let len: usize = rng.gen_range(8..=16);
            let text: String = rng
                .sample_iter(&Alphanumeric)
                .take(len)
                .map(char::from)
                .collect();

            let end_batch = i == items_in_batch - 1; // 最后一条为结束标志
            let req = OutputRequest {
                pid,
                content: format!("P{}-B{}-{}", pid, b + 1, text),
```

```
                end_batch,
            };

            // 发送请求到 SPOOLING 进程
            if tx_req.send(SpoolMessage::Request(req)).is_err() {
                eprintln!("进程 {}: SPOOLING 通道关闭，停止发送。", pid);
                return;
            }

            // 随机调度：随机短暂休眠，模拟各进程随机输出
            let sleep_ms = rng.gen_range(40..=160);
            thread::sleep(Duration::from_millis(sleep_ms));
        }

        // 批次之间也随机暂停
        let pause_ms = rng.gen_range(100..=300);
        thread::sleep(Duration::from_millis(pause_ms));
    }

    println!("进程 {}: 所有批次发送完毕。", pid);
}

fn main() {
    // 通道：用户进程 -> SPOOLING
    let (tx_req, rx_req) = mpsc::channel::<SpoolMessage>();
    // 通道：SPOOLING -> 设备
    let (tx_dev, rx_dev) = mpsc::channel::<OutputBlock>();

    // 启动设备线程
    let device_handle = thread::spawn(move || device_thread(rx_dev));

    // 启动 SPOOLING 输出进程
    let tx_dev_for_spool = tx_dev.clone();
    let spool_handle = thread::spawn(move || spooling_thread(rx_req,
        tx_dev_for_spool));

    // 启动两个产生输出请求的进程
    let tx_req_p1 = tx_req.clone();
    let p1 = thread::spawn(move || producer_thread(1, tx_req_p1, 3));
```

```
    let tx_req_p2 = tx_req.clone();
    let p2 = thread::spawn(move || producer_thread(2, tx_req_p2, 3));

    // 等待两个进程结束
    let _ = p1.join();
    let _ = p2.join();

    // 发送关机信号给 SPOOLING
    let _ = tx_req.send(SpoolMessage::Shutdown);
    // 主动丢弃发送端，促使设备线程在通道关闭后退出
    drop(tx_req);
    drop(tx_dev);

    let _ = spool_handle.join();
    let _ = device_handle.join();

    println!("\n实验6 SPOOLING 模拟结束。");
}
```

5 实验步骤与运行方法

5.1 构建与运行

在 macOS 的 zsh 终端执行：

```
cd "/Users/laijingkang/Documents/study/大三上/操作系统/OS-TASK/task6"
cargo run
```

5.2 预期输出形态

程序将打印多个批次块的消费过程：

```
=== 输出设备：接收批次 1，条目 4 ===
[Batch 1 / Item 1] 来自进程 1: P1-B1-...
[Batch 1 / Item 2] 来自进程 1: P1-B1-...
[Batch 1 / Item 3] 来自进程 1: P1-B1-...
[Batch 1 / Item 4] 来自进程 1: P1-B1-...
=== 批次 1 输出完成 ===
...
实验6 SPOOLING 模拟结束。
```

6 结果分析与讨论

- **顺序性保障**：设备线程只按批次顺序消费，避免在用户进程并发下出现乱序打印。
- **批次完整性**：`end_batch` 明确边界，SPOOLING 按块提交，设备层无需解析细粒度请求。
- **扩展性**：可引入缓冲容量、速率限制与度量收集（队列长度、吞吐、平均等待时间），用于进一步性能评估。

7 结论

本实验完成了 SPOOLING 输出系统的线程/通道实现，验证了在并发场景下以批次块驱动外设可提升吞吐且简化设备端逻辑。后续可参数化生产速率与缓冲大小，并记录度量以研究不同负载下的表现。

8 参考与仓库

项目仓库：https://github.com/shufufufu/OS-TASK