中文摘要

本课程设计面向《操作系统》课程的“进程创建、控制、终止和通信的程序实现”选题，使用 Python 在 Linux 用户态实现一个父进程主程序，能够按需创建与管理多个子进程，并在程序运行时切换三种进程间通信方式（管道通信、直接通信、间接通信），完成“文本输入 → 原子化处理 → 词频统计 → 报告输出”的可组合流水线运行。

系统遵循 Unix 哲学组织 Stage 子程序，父进程负责统一调度与运行控制：按配置文件拼装流水线、逐步启动子进程、读取 /proc 展示 PID/状态/CPU/RSS 等关键指标，并支持向子进程发送 STOP/CONT/TERM/KILL 等信号完成进程生命周期控制。通过对 fork/exec/wait/pipe 等机制的原理分析，并结合运行现象（阻塞、EOF 传播、资源回收等）进行对应说明，报告给出一种稳定可复现的实现方案。

关键词：操作系统；进程管理；进程间通信；管道；信号；/proc；Python

目 录

中文摘要.....

第一章 引言..... 1

1.1 设计目的与意义..... 1

1.2 进程、程序与线程：基本概念..... 1

1.3 UNIX/Linux 进程模型要点..... 1

1.3.1 进程资源视图..... 2

1.4 题目要求..... 2

1.5 设计环境..... 2

1.6 关键机制与理论点概览..... 2

1.7 实现目标..... 3

第二章 需求分析..... 4

2.1 功能性需求..... 4

2.2 通信方式的定义与边界..... 4

2.3 需求到实现的映射..... 4

2.4 非功能性需求..... 5

2.5 验收标准..... 5

2.6 约束条件..... 6

2.7 运行时可选配置：通信方式与流水线..... 6

2.8 典型实验流程..... 6

第三章 概要设计..... 7

3.1 总体设计思路..... 7

3.2 进程组织方式..... 7

3.3 总体结构示意..... 7

3.4 通信模块总体方案 7
3.5 配置驱动与流水线拼装 8
3.5.1 配置字段设计：pipeline.toml 8
3.5.2 Stage 类型系统与链式校验 8
3.5.3 典型 Stage 链示例：词频统计 9
3.5.4 输出目录与归档约定 9
3.5.5 单步运行的时序设计 9
3.6 三种通信方式对比 9
第四章 详细设计与实现 11
4.1 模块划分与职责 11
4.2 设计原则与整体取舍 11
4.3 进程创建与执行模型：fork/exec/wait 对应关系 11
4.3.1 fork 的资源继承与写时复制：COW 12
4.3.2 exec 与 close-on-exec：FD_CLOEXEC 12
4.3.3 文件描述符模型与“字节流”抽象 12
4.3.4 文件描述符重定向与关闭策略 12
4.3.5 stderr 重定向与“数据面/控制面”隔离 13
4.3.6 等待与退出码：避免僵尸进程 13
4.3.7 创建与等待的伪代码：原理对应 13
4.4 进程控制与终止：信号机制 13
4.4.1 信号语义与可捕获性 14
4.4.2 退出码与 wait 状态的解释 15
4.5 三种通信方式的实现 15
4.5.1 三种通信方式的 FD 拓扑：示意 15
4.5.2 pipe：标准输入输出管道 16
4.5.3 direct：socketpair 点对点通信 16

4.5.4 indirect：mailbox 间接通信 16
4.6 核心调度实现：pipeline_runner 17
4.7 Stage 子程序设计：遵循 Unix 哲学 17
4.8 /proc 进程信息展示：用户态观测 18
4.8.1 /proc 关键字段与换算方法 18
4.8.2 /proc/<pid>/stat 的解析注意事项 18
4.9 单步执行与运行中控制 19
第五章 调试与运行 20
5.1 调试思路 20
5.2 实验准备：配置与输入 20
5.3 关键问题与解决 21
5.3.1 管道阻塞与缓冲现象 21
5.3.2 EOF 传播与 FD 关闭 21
5.3.3 SIGPIPE / BrokenPipe 的常见现象 21
5.3.4 终止与回收：僵尸进程 21
5.4 运行结果与现象分析 21
5.4.1 样例与配置设计 22
5.4.2 三种通信方式的对照观察点 22
5.4.3 实验：背压阻塞复现：pipe/direct 22
5.4.4 实验：EOF 传播问题的定位：故障注入 23
5.4.5 Web 控制面：配置摘要与运行面板 23
5.4.6 单步执行：进程信息与输出预览 23
5.4.7 实验：STOP/CONT 控制与状态变化 23
5.4.8 实验：TERM/KILL 终止与回收 24
5.4.9 三种通信方式的可切换性 24
5.4.10 实验结果截图 24

第六章 课设小结 26
6.1 完成情况与效果 26
6.2 创新点与加分项 26
6.3 理论对应与个人收获 26
6.4 改进方向 27
参考文献 28
附录 29

第一章　引言

1.1　设计目的与意义

本课程设计旨在通过“用户态可运行、可交互、可复现实验现象”的方式，帮助理解操作系统中进程（Process）的基本概念、生命周期与调度可观测性，以及管道、信号与进程间通信（IPC）的典型实现机制 [1, 2, 3, 4]。相较于只讨论抽象概念，本设计将理论点落实到可验证的程序行为：创建子进程、控制其暂停/继续、终止与回收，并以不同通信机制完成同一业务流程，从而建立“现象—机制—原理”的对应关系。

本设计选择“文本词频统计”作为业务载体，是因为其数据流特征清晰、阶段可原子化拆分，适合在 Unix 哲学指导下构建可组合流水线。以此为例，可以稳定展示“输入 $\rightarrow$ 处理 $\rightarrow$ 输出”的完整闭环，并围绕进程创建与 IPC 的共性机制展开分析。

1.2　进程、程序与线程：基本概念

从操作系统视角看，“程序”是静态的可执行文件（或脚本 + 解释器的组合），而“进程”是程序的一次运行实例，拥有独立的地址空间与内核可管理的资源集合（打开文件、信号处理、环境变量、调度实体等） [1, 2]. 线程（Thread）通常被视为进程内部的执行流；同一进程内多个线程共享地址空间与大部分资源，但线程拥有各自的寄存器上下文与栈。

为了突出本课设“进程创建/控制/终止/通信”的要求，本设计以多进程作为基本并发与隔离单元：父进程负责控制与观测，子进程负责数据处理。表 1 对相关概念做出对比，作为后续章节讨论的基础。

表 1　程序、进程与线程概念对比

概念	是否运行态	资源边界	典型用途
程序	否	文件系统中的静态对象	作为可执行载体被装载运行
进程	是	独立地址空间 + FD + 信号等	隔离与资源控制、IPC、作业管理
线程	是	共享进程资源；独立寄存器/栈	同进程内并发、降低切换开销

1.3　UNIX/Linux 进程模型要点

UNIX/Linux 采用“父子进程树”的组织方式：除内核进程外，用户态进程通常由已有进程创建而来，形成以 init/systemd 为根的进程树 [5, 6]。父进程通过等待机制（wait/waitpid）回收子进程的退出状态；若父进程没有回收，则子进程可能以僵尸态（Z）保留在进程表中，直到被回收 [7, 8, 1].

1.3.1 进程资源视图

进程在内核中的关键可观测资源可抽象为：地址空间、文件描述符表、信号处理方式以及调度相关信息等。图 1 给出简化的资源视图，用于解释本设计为何能通过“FD 连接 + 信号 + wait 回收”完成题目要求。

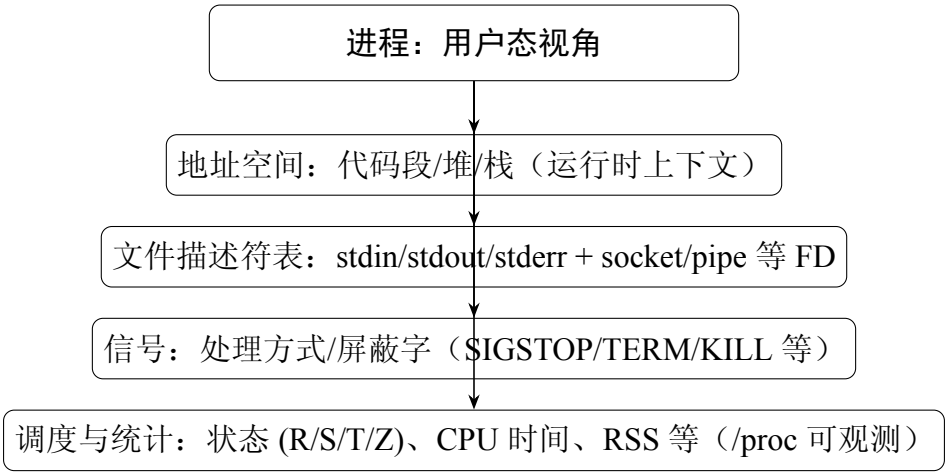


图 1　进程资源视图：地址空间、FD、信号与调度信息

1.4 题目要求

本课设选题为“进程创建、控制、终止和通信的程序实现”。题目要求使用高级语言实现 3～4 个应用程序，其中一个为主程序（父进程），负责按需创建/终止其他进程；并支持父子进程或子进程之间的通信，通信方式必须包含直接通信、间接通信或管道通信，且在程序运行时可选。

1.5 设计环境

实验环境为 Linux + Python 3（用户态）。系统通过操作系统提供的进程与 IPC 接口实现所需功能，包括：创建子进程并执行独立程序、通过文件描述符传递字节流、通过信号控制进程状态、以及通过等待机制回收子进程退出状态等 [7, 8, 9].

为便于观察进程状态与资源占用，系统读取 Linux 的 /proc 虚拟文件系统展示关键字段（PID、PPID、状态、CPU 时间、RSS、启动时间等），并在每个 Stage 运行时输出快照以支撑后续章节的原理对应分析。

1.6 关键机制与理论点概览

本设计虽使用 Python 标准库实现，但其核心行为可直接映射到 Unix/Linux 的经典机制：进程创建与装载（fork/exec）、等待与回收（wait/waitpid）、FD 与重定向

（pipe/dup2/socketpair）、信号（kill）以及 /proc 观测等。表 2 给出“实现点—理论点”的概览，便于后文展开分析。

表 2　关键机制与理论点概览

机制/接口	在本设计中的用途	对应理论点：章节
fork/exec（或 posix_spawn）	创建并执行 Stage 子程序	进程创建与装载模型：第 4 章
wait/waitpid	回收退出状态、防止僵尸	生命周期闭环与僵尸原理：第 4/5 章
pipe、stdin、stdout	经典 pipeline 串联	管道背压、EOF 传播：第 4/5 章
socketpair（Unix domain socket）	点对点“直接通信”	FD 抽象与直接通信拓扑：第 4 章
信号（kill）	STOP/CONT/TERM/KILL 控制	信号语义与状态流转：第 4/5 章
/proc 虚拟文件系统	展示状态/CPU/RSS/启动时间	用户态可观测性与调度概念：第 4/5 章

1.7　实现目标

结合题目要求与运行可操作性，系统的实现目标包括：

- 至少一个父进程主程序 + 多个子进程 Stage（每个 Stage 为独立入口文件）；
- 父进程可展示子进程信息，并可发送 STOP/CONT/TERM/KILL 等信号控制其生命周期；
- 运行时可切换三种通信方式：pipe：管道，direct：socketpair 点对点，indirect：邮箱进程中转；
- 在同一组输入样例上，三种通信方式均可稳定完成词频统计并输出报告。

第二章　需求分析

2.1　功能性需求

围绕题目要求与运行可操作性，系统需满足以下功能：

1. 父进程主程序：提供控制面，能够创建、观察、控制与终止子进程；
2. 多 Stage 流水线：支持选择一个或多个 Stage 串联运行，以完成词频统计任务；
3. 三种通信方式：同一套业务流程可切换管道通信、直接通信与间接通信，并在运行时选择；
4. 进程信息展示：运行时可展示每个 Stage 的进程关键信息（/proc 快照），用于说明“用户态进程管理”；
5. 过程可控：支持暂停/继续/终止/强制终止等信号控制，覆盖进程生命周期的关键阶段。

2.2　通信方式的定义与边界

题目要求包含“直接通信、间接通信或管道通信”，且通信方式在程序运行时必须可选。为避免概念混淆，本设计对三类通信方式的边界定义如下：

- 管道通信（pipe）：通过标准输入/输出或匿名管道在进程间传递字节流，形成线性流水线；
- 直接通信（direct）：发送方与接收方之间存在点对点连接（本设计使用 socketpair 建立 Unix domain socket 直连）；
- 间接通信（indirect）：发送方与接收方之间不直接相连，而是通过“邮箱/中介”进程进行转发，体现解耦思想。

需要说明的是：本课程设计聚焦进程与 IPC 的机制展示，因此统一采用“字节流”作为通信数据单元；不引入共享内存、消息队列等更复杂的 IPC 形态，以降低实现复杂度并保证运行过程稳定。

2.3　需求到实现的映射

为避免“功能点堆砌而缺少对应说明”，本设计将题目要求拆分为若干可验收条目，并在实现中逐一落地。表 3 给出“题目要求—实现要点”的对应关系。

表 3　题目要求与实现要点映射

题目要求	实现要点
创建进程	父进程按配置逐步启动 Stage 子进程；每个 Stage 为独立入口文件；展示 PID/PPID 等信息。
控制进程	父进程可向子进程发送 STOP/CONT/TERM/KILL 信号并观察状态变化（/proc state）。
终止与回收	先发送 SIGTERM 并等待；超时未退出则发送 SIGKILL；随后 wait 回收退出状态，避免僵尸进程。
管道通信	通过 stdin/stdout 管道串联数据流；正确关闭 FD 以保证 EOF 传播。
直接通信	通过 Unix domain socket 的 socketpair 点对点连接相邻进程，字节流直连。
间接通信	引入“邮箱/中介进程”转发字节流，实现发送方与接收方解耦。
运行时可选	通信方式由配置文件字段指定，父进程在运行前读取并按模式启动。

2.4　非功能性需求

1. 运行稳定：避免阻塞死锁与僵尸进程，异常情况下应尽力清理资源；
2. 交互清晰：控制面信息明确，步骤可复现；
3. 依赖最小：仅使用 Python 标准库与 Linux 用户态接口；
4. 配置驱动：通过配置文件统一管理输入、通信方式与流水线组成，减少手工输入错误。

2.5　验收标准

结合题目要求与课程报告关注点，本设计给出如下可验收标准：

1. 进程创建：父进程能够按需启动多个子进程，并展示其 PID/PPID；
2. 进程控制：父进程能够向子进程发送 STOP/CONT/TERM/KILL 并观察到状态变化；
3. 进程终止：支持优雅终止与强制终止，且不遗留僵尸进程；
4. 三种通信方式：pipe/direct/indirect 均可运行时选择，并完成相同业务闭环；
5. 结果一致性：在相同输入与 Stage 链下，不同通信方式得到一致的词频统计输出；
6. 可观测性：能通过 /proc 展示关键指标，并在单步运行时输出快照。

2.6 约束条件

系统运行环境固定为 Linux + Python 3。父进程与子进程均以用户态方式实现，不涉及内核态开发。

2.7 运行时可选配置：通信方式与流水线

为降低人工配置差错，系统采用配置驱动方式：输入文件、通信方式、Stage 顺序与参数均由 TOML 配置文件统一描述。父进程每次运行前重新读取配置，从而在相同业务逻辑下实现“切换通信方式与流水线”的实验对照。

2.8 典型实验流程

为便于形成“现象—机制—原理”的对应关系，本设计给出表 4 所示的典型实验流程。该流程覆盖题目要求的“创建、控制、终止、通信可选”四个关键点。

表 4 典型实验流程

步骤	操作	预期现象（对应原理）
1	启动一次运行并查看配置摘要	父进程读取配置文件并完成参数解析。
2	进入单步执行，启动第一个 Stage	子进程 PID/PPID 可观测；/proc 可读到状态与资源统计。
3	运行中发送 SIGSTOP/SIGCONT	子进程状态在运行/睡眠与停止 (T) 之间切换（信号机制）。
4	切换 run.mode 为 direct/indirect 并重跑	连接拓扑变化；间接模式出现 mailboxd 中介进程。
5	发送 SIGTERM，必要时 SIGKILL，并回收退出状态	对比优雅终止与强制终止；wait 回收避免僵尸进程。

第三章　概要设计

3.1　总体设计思路

系统采用“父进程控制面 + 多子进程 Stage”的组织方式：父进程负责读取配置、拼装流水线、创建并管理子进程，以及在运行过程中展示进程信息与提供控制指令；子进程（Stage）仅完成单一数据处理职责，通过标准输入/输出或 socket 字节流与上下游连接形成处理链。

3.2　进程组织方式

父进程作为唯一交互入口，按配置逐步启动 Stage 子进程。运行时可选三种通信方式：管道通信、直接通信与间接通信。间接通信通过额外的“邮箱/中介”进程实现中转转发，使发送方与接收方在结构上解耦。

3.3　总体结构示意

图 2 给出系统的总体结构。父进程负责控制与观测；数据在 Stage 之间以字节流形式向前传递。对于间接通信模式，父进程还会创建额外的 mailbox 进程用于转发（在结构上相当于“消息中介”）。

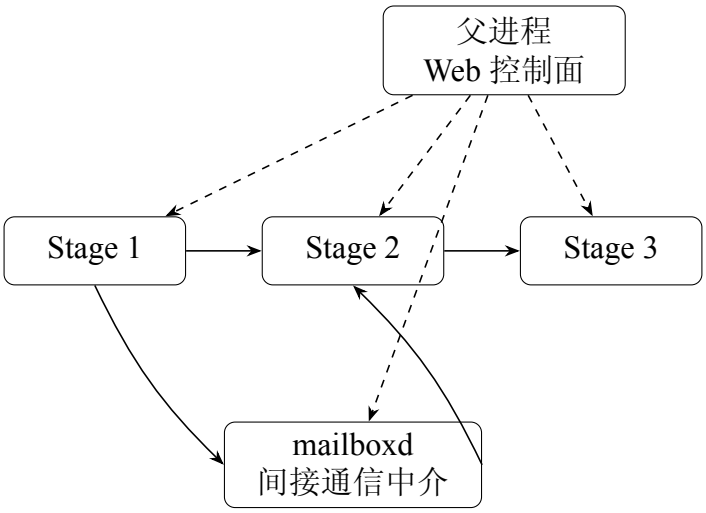


图 2　系统总体结构：控制流与数据流分离

3.4　通信模块总体方案

三种通信方式在接口层保持一致：对 Web 控制面来说，均表现为“父进程启动子进程并向其输入字节流，读取其输出字节流”。不同点在于底层连接方式（pipe / socketpair / mailbox 转发）与 FD 管理策略。

3.5 配置驱动与流水线拼装

系统通过 TOML 配置文件指定输入文件、通信方式与 Stage 列表。父进程在运行前进行合法性校验（例如 Stage 输入/输出类型兼容性），并将部分参数按约定注入到 Stage 命令行，降低配置复杂度。

3.5.1 配置字段设计：pipeline.toml

本设计采用“配置驱动”是为了降低操作失误概率：运行参数、输入文件与流水线顺序均写入配置文件，运行时只需修改少量字段即可复现实验现象。表 5 给出主要字段说明与设计理由。

表 5 pipeline.toml 关键字段说明

字段	含义	设计理由
input.path	输入文本路径	输入作为 Source 的参数，由父进程统一注入，避免 Stage 与路径绑定。
run.mode	通信方式（pipe/direct/indirect）	满足“运行时可选”要求；同一业务流程切换通信机制。
run.outputs_dir	输出目录	统一落盘，便于展示“每步输出 + 最终报告 + 日志”。
run.default_topn	默认 TopN	报告类 Stage 的默认参数，减少配置负担。
pipeline.stages	Stage 链（线性数组）	顺序即执行顺序;可通过注释禁用单个 Stage。
stage_args.*	Stage 结构化参数	参数与顺序解耦，拖拽/调整顺序时不易写错。

3.5.2 Stage 类型系统与链式校验

为体现“可组合流水线”的工程约束，本设计为 Stage 定义了数据类型：lines（行流）、tokens（词元流）、counts（计数流）、report（报告文本）。父进程在运行前校验相邻 Stage 的输入/输出类型，避免出现“上游输出为 tokens，但下游期望 lines”之类的组合错误。

这种类型校验的意义在于：即使调整 Stage 顺序，也能在启动前给出可解释的错误提示，而不是运行到一半才失败，从而提升稳定性。此外，debug.delay 之类的透传 Stage 可声明为“任意类型输入/任意类型输出”，用于复现控制信号与阻塞现象。

3.5.3 典型 Stage 链示例：词频统计

为体现“遵循 Unix 哲学的程序串联”，表 6 给出一个典型词频统计流水线的 Stage 链示例。每个 Stage 只完成一种原子操作（source / tokenize / filter / count / report），并通过类型系统保证链式组合合法。

表 6　典型 Stage 链示例：词频统计

Stage ID	输入类型	输出类型	职责（原子操作）
source.file	-	lines	读取输入文件，输出行流
tokenize.en_words	lines	tokens	提取英文单词并规范化为小写
token.len_range	tokens	tokens	按长度区间过滤 token
count.tokens	tokens	counts	统计词频（TSV：token<TAB>count）
report.text	counts	report	生成文本报告并落盘

3.5.4 输出目录与归档约定

为了让运行结果可追溯，父进程将中间结果与日志统一落盘。常见输出包括：每步输出文件、最终报告文件与统一 stderr 日志文件。该设计便于在第 5 章对照分析时引用具体文件路径，并可直接打开预览。

3.5.5 单步运行的时序设计

本设计提供“按 Stage 单步执行”的运行模式。其核心时序为：父进程启动子进程 → 建立通信 FD→ 发送上游输出（或文件输入）→ 读取子进程输出到文件 →wait 回收退出码 → 展示 /proc 快照与输出预览。图 3 给出单步运行的简化时序图。

3.6 三种通信方式对比

从操作系统抽象看，三种通信方式本质上都是“通过文件描述符传输字节流”，差异体现在连接方式、耦合程度与可观测性上。表 7 对三种方式做出对比。

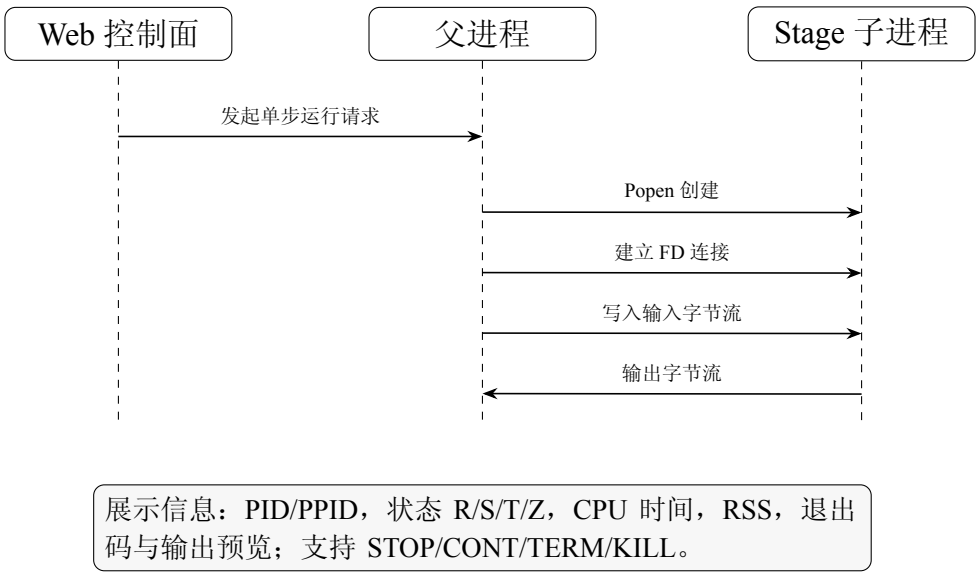


图 3　单步运行时序：父进程作为控制面与 I/O 汇聚点

表 7　三种通信方式对比

模式	连接机制	进程关系	特点（适用场景）
pipe	stdin/stdout 管道	相邻 Stage 串联	Unix pipeline 经典模式；实现简单；需注意缓冲与 EOF 传播。
direct	socketpair 点对点	相邻 Stage 直连	全双工字节流；比 pipe 更通用；仍属于直接通信，点对点。
indirect	mailbox 中转转发	增加中介进程	发送方与接收方解耦；可扩展为多路路由；实现需管理额外进程与 FD 传递。

第四章　详细设计与实现

4.1　模块划分与职责

为保证实现清晰且便于维护，系统按单一职责拆分为若干核心模块，见表 8。

表 8　核心模块划分与职责

模块	职责与对应文件
父进程入口与控制面（Web）	运行入口、配置展示、单步执行与信号控制：src/main.py、src/web_main.py
配置解析与校验	从 TOML 读取输入、通信方式与流水线，完成基本校验：src/config_loader.py
Stage 注册与类型约束	维护 Stage ID 与入口脚本映射，校验链式组合的输入/输出类型：src/stage_registry.py
进程与通信核心	封装三种通信方式的底层实现，提供统一的运行与控制接口：src/pipeline_runner.py
间接通信中介	mailbox 进程按路由转发字节流，体现间接通信：src/mailboxd.py
进程信息读取	读取 /proc 并格式化展示关键进程信息：src/procfs.py

上述拆分体现“控制面与数据面分离”的思想：父进程负责创建与管理进程并汇聚 I/O，子进程负责数据处理。Stage 设计遵循 Unix 哲学，以可组合的标准输入输出接口组织处理链；进程管理与通信复杂度集中在父进程核心模块中。

4.2　设计原则与整体取舍

本课程设计以清晰性、可复现性与可观测性为优先目标。实现上采用单一职责的 Stage 子程序，统一以字节流作为 I/O 抽象；父进程集中封装进程管理与 IPC，避免在控制面重复实现关键逻辑；运行过程输出 /proc 快照与退出码，便于将实验现象与机制对应；同时对阻塞、EOF 传播与僵尸回收等问题提供防御性处理以保证整体稳定。

4.3　进程创建与执行模型：fork/exec/wait 对应关系

Unix/Linux 的典型进程创建与执行模型为 fork–exec：fork 创建子进程（继承父进程的打开文件描述符、环境变量等），exec 装载新程序映像，wait/waitpid 负责等待并回收子进程退出状态，避免产生僵尸进程 [7, 8, 9].

本文使用 Python 的 subprocess 启动 Stage 子进程。父进程使用 subprocess.Popen 运行对应 Stage 脚本。

在 POSIX 系统上，该接口通常基于 fork/exec（或 posix_spawn）完成子进程创建与程序装载 [9, 8]。

因此，从操作系统原理角度看，本项目仍可用 fork、exec、wait 的经典模型解释。父进程在创建子进程前准备所需的文件描述符（stdin/stdout 或 socket），随后启动子进程执行对应 Stage 脚本；子进程退出后，父进程通过等待机制获取退出码，完成生命周期闭环。

4.3.1 fork 的资源继承与写时复制：COW

在 fork 语义下，子进程会继承父进程的许多运行时上下文：打开的文件描述符、环境变量、当前工作目录、信号处理方式等；地址空间通常通过写时复制（Copy-on-Write, COW）实现，以降低创建开销 [1, 2, 8]。在本设计中，这一继承特性主要体现为：其一，FD 继承：父进程在创建子进程前构造好的 pipe、socket 等 FD，会在子进程中可见，从而实现“启动即连通”；其二，exec 后 FD 默认仍可保留：exec 替换程序映像，但除非设置 close-on-exec，否则不会自动关闭已打开的 FD，因此必须制定明确的 FD 管理策略。

4.3.2 exec 与 close-on-exec：FD_CLOEXEC

为避免 FD 泄露，POSIX 提供 close-on-exec 标志（FD_CLOEXEC）：当进程执行 exec 时，标记为 close-on-exec 的 FD 会被自动关闭 [9, 7]。在本设计中，父进程在启动子进程时采取“默认关闭 + 显式传递”的策略：对 Stage 子进程使用 close_fds 关闭非必要 FD；对 mailboxd 进程则使用 pass_fds 明确保留路由所需的 FD。该策略将“继承的便利”与“资源边界可控”结合起来，是保证间接通信稳定的关键实现点之一。

4.3.3 文件描述符模型与“字节流”抽象

在 UNIX 中，大多数 I/O 与 IPC 都可抽象为“通过文件描述符读写字节流”。进程持有的 FD 是一个小整数索引，指向内核中的打开文件对象（open file description）。管道与 socketpair 也以 FD 形式暴露给用户态，因此可以像文件一样读写 [7, 8, 9]。

本设计统一将三种通信方式抽象为“向下游写入字节流、从上游读取字节流”，使得在上层逻辑上可以切换通信机制而不改变业务 Stage 的实现方式。图 4 给出简化的 FD 视图，说明为何 pipe、socket、mailbox 在程序结构上可以统一处理。

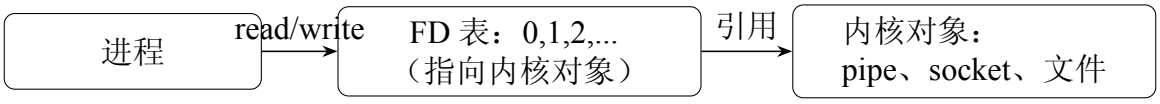


图 4　文件描述符抽象：进程通过 FD 访问内核对象

4.3.4 文件描述符重定向与关闭策略

管道与 socketpair 在内核中都以“可读/可写端”的方式呈现给用户态进程，用户态通过 FD 访问该内核对象。父进程在启动子进程时为其设置 stdin/stdout 的 FD，即可把

子进程的标准输入输出连接到指定通道。

需要强调的是：EOF 的传播依赖于“所有写端都被关闭”。因此父进程在建立连接后必须及时关闭自己持有的多余 FD 引用，否则下游进程可能永远收不到 EOF 而阻塞。这一点在第 5 章调试现象中会进一步结合实验说明。

4.3.5 stderr 重定向与“数据面/控制面”隔离

为保持流水线数据流的纯净性，Stage 约定“数据走 stdout，日志走 stderr”。父进程将子进程的 stderr 统一重定向到日志文件，避免日志输出干扰控制面展示与数据预览；同时也便于在调试章节中复现与分析异常信息。该做法符合 Unix 哲学下的常见约定，有助于形成可组合流水线 [7].

4.3.6 等待与退出码：避免僵尸进程

子进程退出后，内核会保留其部分信息（如退出码）直到父进程执行 wait 回收；若父进程从不 wait，则该子进程将长期处于僵尸态（Z）。本设计在单步执行与整条流水线执行两条路径上，都确保父进程在合适时机对每个子进程执行等待并获取退出码，从而满足题目对“终止与资源回收”的要求 [7, 1].

4.3.7 创建与等待的伪代码：原理对应

下面以“父进程启动一个 Stage 并回收退出码”为例，用伪代码刻画 fork/exec/wait 的关键步骤（省略异常处理细节）：

算法 1: 创建 Stage 进程并回收退出状态

输入: stdin_fd、stdout_fd、stderr_fd；待执行程序 stage_program 及参数 argv、环境 env
输出: 子进程退出状态（exit code 或终止信号）

```
准备 stdin/stdout/stderr 对应的 FD（pipe 或 socket）
pid ← fork()
if pid == 0 then
    dup2(stdin_fd, 0); dup2(stdout_fd, 1); dup2(stderr_fd, 2)
    execve(stage_program, argv, env)
else
    关闭父进程中不再需要的 FD 引用
    waitpid(pid) 并读取退出状态（避免僵尸）
```

4.4 进程控制与终止：信号机制

信号是 Unix 中典型的异步通知机制，可用于进程控制与异常处理 [7, 8, 9]。本设计在 Web 控制面提供 STOP/CONT/TERM/KILL 等控制操作，分别对应 SIGSTOP、SIGCONT、

SIGTERM、SIGKILL，并可通过 /proc 观察到状态变化与资源统计的差异。

图 5 给出本设计涉及的典型状态流转：运行态进程可被 SIGSTOP 暂停，SIGCONT 恢复；终止后若父进程未 wait 回收，则进入僵尸态，直到被回收。

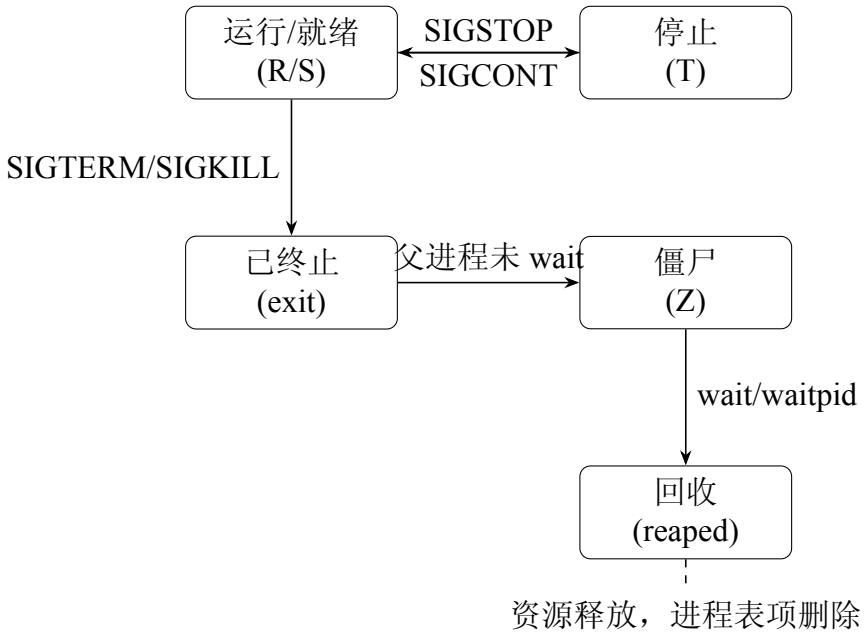


图 5　典型进程状态流转：信号控制与 wait 回收

为保证稳定性，父进程终止一条流水线时采用“先发送 SIGTERM 并等待短超时，若仍未退出则发送 SIGKILL，最后 wait 回收所有子进程”的策略，避免僵尸进程遗留。

4.4.1　信号语义与可捕获性

为了将“控制命令”与“操作系统语义”对应起来，需要强调信号的两类差异：其一是信号是否可被进程捕获/忽略；其二是信号到达后在内核中引发的状态变化 [7, 8, 9]。表 9 总结本设计涉及的典型信号。

表 9　本设计涉及的典型信号语义

信号	作用	可捕获/忽略	对应可观测现象
SIGSTOP	强制停止进程	否/否	/proc 状态变为 T；CPU 时间停止增长。
SIGCONT	恢复停止进程	-	/proc 状态从 T 恢复为 R/S；继续执行。
SIGTERM	请求优雅终止	是/是	进程可清理资源后退出；退出码由 wait 获取。
SIGKILL	强制终止进程	否/否	立即退出；父进程仍需 wait 回收避免僵尸。

4.4.2 退出码与 wait 状态的解释

在 Unix 中，父进程通过 wait/waitpid 获取子进程的退出状态：正常退出时携带应用层退出码；被信号终止时携带“由哪个信号导致终止”的信息 [7, 8]。本项目在控制面展示 Python subprocess.Popen.returncode，其常见解释规则为：

- returncode >= 0：子进程调用 exit(returncode) 正常退出；
- returncode < 0：子进程被信号终止，-returncode 为信号编号（例如 -9 表示 SIGKILL）。

该规则使得 SIGTERM 与 SIGKILL 的差异可以直接反映为不同的 returncode，再结合 /proc 状态与 wait 回收，形成可观测的完整闭环。

4.5 三种通信方式的实现

系统实现三种可切换通信方式：pipe（管道）、direct（socketpair 直连）、indirect（mailbox 中转）。三种方式均以“字节流”作为统一抽象接口，从而可在运行时切换而不改变业务处理链。

4.5.1 三种通信方式的 FD 拓扑：示意

图 6 给出三种模式下相邻两个 Stage 的连接拓扑示意。可以看到三者本质上都是“把某个 FD 作为对方的 stdin/stdout”，区别在于是否引入中介进程，以及 FD 在哪些进程间被持有。

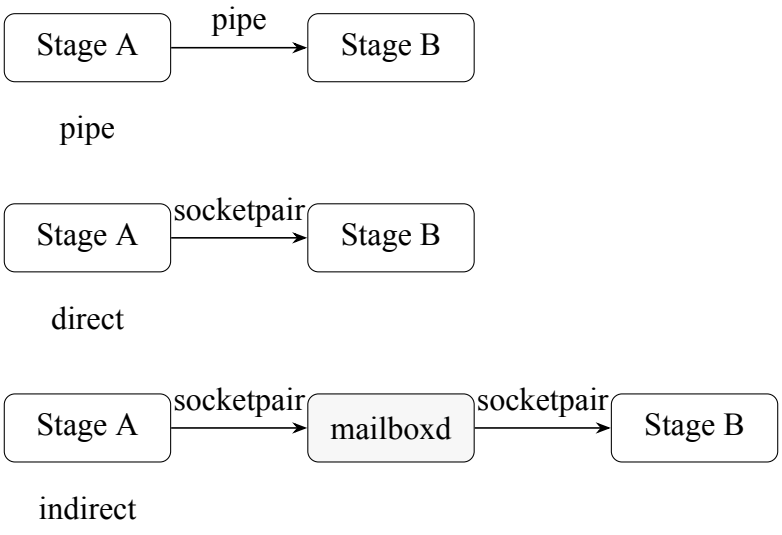


图 6　三种通信方式的连接拓扑：相邻 Stage 示例

4.5.2 pipe：标准输入输出管道

管道通信是 Unix pipeline 的经典用法：上游进程的 stdout 连接到下游进程的 stdin，实现单向字节流传递 [7, 8, 9]。本设计在父进程中按 Stage 顺序创建子进程，并将前一 Stage 的 stdout 作为下一 Stage 的 stdin。为了避免管道写满导致上游阻塞，父进程对最终输出端启动读取线程及时读取数据。

关键点：关闭多余 FD，保证 EOF 传播 在多级管道中，父进程通常会短暂持有多条管道的读端/写端。如果父进程没有及时关闭不再需要的写端引用，下游进程将无法收到 EOF，从而表现为“最后一步一直等待输入”。这类现象在第 5 章会结合截图占位与对照表说明。

关键点：避免管道写满导致死锁 当父进程不读取最终输出，而最后一个 Stage 继续向 stdout 写数据时，stdout 所在的 pipe 缓冲区可能写满，导致最后一个 Stage 阻塞；进一步会让整个流水线停滞。为避免该问题，本设计为最终输出启动读取线程持续读取数据。

4.5.3 direct：socketpair 点对点通信

直接通信模式使用 Unix domain socket 的 socketpair 在两个相邻进程之间创建点对点连接 [10, 8]。与管道相比，socketpair 提供全双工通道；在本设计中仍将其作为“单向数据流”使用，以保持流水线接口一致，并突出“点对点直接通信”的概念：发送方与接收方之间存在直接连接，数据不经过第三方中介。

为何选择 Unix domain socket 相较于 TCP/UDP，Unix domain socket 不依赖网络栈与端口号，通信仅发生在本机内核中，创建与使用方式与 FD 模型一致，适合用于讲解“点对点直接通信”的 OS 机制，同时保持实现简单（无需协议设计）。

4.5.4 indirect：mailbox 间接通信

间接通信模式引入额外的 mailbox 进程（mailboxd）作为中介，使发送方与接收方在结构上解耦。父进程为相邻 Stage 创建两对 socket：mailboxd 的职责被刻意设计得非常单一：不解析内容，仅按路由表从输入 FD 读字节并写入输出 FD。父进程启动 mailboxd 时需要显式传递它将使用的 FD（并关闭父进程自身持有的多余引用），从而保证资源边界清晰、EOF 语义正确。

FD 传递与 pass_fds 在默认安全策略下，父进程启动子进程会关闭大部分非标准 FD，以避免句柄泄露。本设计在启动 mailboxd 时显式列出需要保留的 FD（即各条路由的 in/out 端），让 mailboxd 能访问这些 FD 并完成转发。这一实现细节与“FD 继承/关闭策略”直接相关，也是间接通信区别于直接通信的重要点。

mailboxd 的并发模型 mailboxd 采用“每条路由一个线程”的实现方式：每个线程循环读取输入 FD 并写入输出 FD，直到读到 EOF 或收到终止信号。该实现并非追求极致性能的事件驱动模型，但实现路径清晰，便于对应“间接通信引入中介进程”的结构性特征。

4.6 核心调度实现：pipeline_runner

核心调度模块的职责是把“配置中的 Stage 列表”落实为一组真实的子进程，并在运行期间提供可观测与可控制的接口。其实现要点可概括为：

1. 启动前校验：根据 Stage 注册表检查启用项、拼装命令行并做输入/输出类型链式校验；
2. 统一启动：根据 run.mode 选择不同的 FD 连接策略（pipe / socketpair / mailbox）；
3. 输出汇聚：后台线程持续读取最后一个 Stage 的 stdout，避免管道缓冲写满导致阻塞；
4. 生命周期闭环：提供信号广播与 terminate_all，并在退出路径上执行 wait 回收；
5. 单步执行：提供 run_stage_step，将每个 Stage 的输出落盘并返回预览与 /proc 快照用于控制面展示。

从操作系统原理角度看，调度模块最关键的是“正确处理 FD 与 wait”：FD 处理不当会导致 EOF 不传播、进程阻塞；wait 缺失会导致僵尸进程。下面给出 start_pipeline 的抽象化流程（省略异常分支）：

算法 2: start_pipeline 抽象流程

输入: 配置文件；Stage 列表 stages；通信方式 mode；输入路径 input_path；日志路径 log_path
输出: 运行实例 PipelineRun（用于展示、控制、终止与回收）

```
读取配置并构建 Stage 命令列表
根据 mode 建立连接（pipe / socketpair / mailbox）
foreach stage ∈ stages do
    启动子进程：Popen(stage, stdin=..., stdout=..., stderr=log, close_fds=True)
    父进程关闭不再需要的 FD 引用，保证 EOF 能正确传播
启动读取线程持续读取最后一个 stdout，避免缓冲区写满导致阻塞
return PipelineRun
```

4.7 Stage 子程序设计：遵循 Unix 哲学

Stage 子程序由父进程创建并执行，遵循 Unix 哲学：一个 Stage 只做一件事，通过标准输入输出与其他 Stage 组合。为保证在三种通信方式下均可复用，本设计对 Stage 的约定如下：

- I/O 约定：从 stdin 读取文本流（行或 token）；向 stdout 输出处理结果；日志与错误信息输出到 stderr；
- 纯数据面：Stage 不负责进程管理与 IPC 选择，只关注自身的 map/filter/reduce 逻辑；
- 可组合性：输入/输出类型在注册表中显式声明，由父进程在启动前做链式校验；
- 可诊断性：参数错误返回非 0 退出码，并在 stderr 输出可解释信息。

以计数流为例，Stage 之间的数据以 TSV 表示：每行 token<TAB>count。选择 TSV 的原因是：格式直观、无需额外依赖，并与常见的文本处理生态兼容，便于直接打开中间结果文件进行核对。

4.8 /proc 进程信息展示：用户态观测

Linux 的 /proc 虚拟文件系统向用户态暴露了丰富的进程运行信息。本设计读取 /proc/<pid>/stat、/proc/<pid>/cmdline 等文件，提取 PID、PPID、状态、CPU 时间、RSS、启动时间等字段，并在单步运行时输出快照。该做法便于将“信号控制/进程状态/资源占用”与机制对应起来 [8, 11, 12].

4.8.1 /proc 关键字段与换算方法

以 /proc/<pid>/stat 为例，其中包含进程状态、父 PID、CPU 时间（utime/stime，以时钟滴答计）以及启动时间（自系统启动以来的滴答数）等信息。为了将其转换为“可读且可讲解”的指标，本设计进行如下换算：

- CPU 时间（秒）：(utime + stime) / CLK_TCK；
- RSS（字节）：rss_pages * PAGE_SIZE；
- 启动时间（近似绝对时间）：btime + starttime/CLK_TCK，其中 btime 来自 /proc/stat。

这些字段的意义与“调度可观测性”直接相关：通过 STOP/CONT 信号控制可以观察 state 变化，通过长时间运行可观察 CPU 时间增长，通过不同通信方式对比可观察资源占用差异。

4.8.2 /proc/<pid>/stat 的解析注意事项

stat 文件中的第二个字段 comm 被括号包裹，且其内容理论上可能包含空格；因此不能简单用空格切分整行再按下标取值，否则容易在极端情况下解析错位 [8]。本设计在实现上优先定位括号范围后再解析其余字段，并在读取失败时以“不可用”占位，以避免观测模块异常影响主流程。

4.9 单步执行与运行中控制

本设计提供单步执行模式：每次只启动一个 Stage 子进程，父进程作为其上游输入提供者与下游输出接收者。在子进程运行期间，控制面可发送 STOP/CONT/TERM/KILL，并随时刷新 /proc 快照与预览当前输出文件。

第五章　调试与运行

5.1　调试思路

本系统的调试重点在于：避免由于缓冲与 FD 引用导致的阻塞；保证 EOF 能正确传播；在异常退出时仍能正确终止并回收子进程，避免僵尸进程；以及让运行现象能够映射到操作系统原理（信号、等待、管道与 socket 的行为）。

为保证定位过程可复现，调试策略采用分层视角：

1. I/O 层：确认 stdin、stdout、stderr 的连接关系与关闭时机是否符合预期，避免多余写端导致 EOF 不到达；
2. 缓冲层：确认是否存在运行时缓冲导致的“下游等待、上游写满”现象，必要时使用无缓冲输出；
3. 生命周期层：确认异常路径是否执行 wait 回收，避免僵尸；终止采用“先 SIGTERM，超时后 SIGKILL，最后 wait 回收”的策略并可稳定收敛；
4. 可观测层：用 /proc 与日志文件固化现象证据，形成“现象—机制—原理”的对应关系。

5.2　实验准备：配置与输入

为保证实验现象可复现，本设计将运行参数集中在 pipeline.toml 中。实验准备通常包括：

1. 选择输入文件（input.path）；
2. 选择通信方式（run.mode）；
3. 选择 Stage 链（pipeline.stages），必要时插入 debug.delay 以拉长运行时间便于发送信号；
4. 设置输出目录（run.outputs_dir），便于查看每步落盘结果与日志文件。

为了便于“控制信号”实验，可在 Stage 链中临时插入 debug.delay（例如延迟 2 秒）。这样在 Stage 运行期间可稳定执行 STOP/CONT/TERM/KILL 并观察 /proc 中的状态变化。

此外，为避免“实验被输出太短一闪而过”，可选择行数较多的输入文本或适当调大 debug.delay.seconds。本项目的输出会按步骤落盘，便于截取控制面截图与输出文件内容等证据。

5.3 关键问题与解决

5.3.1 管道阻塞与缓冲现象

在流水线中，上游进程向下游写入字节流。当下游消费速度较慢、或父进程没有及时读取最终输出时，内核管道缓冲区可能被写满，上游进程的写操作将阻塞。这类“看似卡住”的现象常见于管道/FD 的背压机制，而非程序逻辑错误 [7, 8].

本设计在父进程侧为最终输出启动读取线程，持续读取最后一个 Stage 的 stdout，从而及时释放缓冲区；同时对子进程设置无缓冲输出，减少“输出不刷新导致下游等待”的现象。

复现实验与对应解释 将 debug.delay 插入到流水线中（例如每行延迟 0.5～2 秒）可以稳定复现“下游慢、上游阻塞”的背压现象：当上游写入速度长期大于下游读取速度时，管道缓冲逐步被占满，上游会在 write 系统调用处睡眠等待缓冲可用。此时 /proc 状态常表现为睡眠态（S），并非 CPU 忙等；恢复下游读取后上游会继续推进。

5.3.2 EOF 传播与 FD 关闭

EOF 是否能到达下游，与“是否还有任何写端 FD 被打开”直接相关。调试过程中若出现某个 Stage 一直等待输入、无法结束，通常是父进程或其他进程仍持有写端 FD 引用。为此，本设计在建立连接后尽快关闭父进程持有的多余端点，并在异常路径中做清理，保证 EOF 能按预期传播。

5.3.3 SIGPIPE / BrokenPipe 的常见现象

当下游进程提前退出（例如被 SIGKILL 终止）而上游仍持续写入时，写端可能收到 SIGPIPE，默认行为会导致进程终止；在高级语言运行时中，这一行为常表现为写入时报错（如 BrokenPipeError）[7, 8]。本设计的处理原则是：让错误尽量在日志中可见，并由父进程统一执行终止与回收，避免实验过程因单个 Stage 异常而留下“半条流水线”。

5.3.4 终止与回收：僵尸进程

若父进程仅发送终止信号而不执行 wait 回收，子进程退出后会进入僵尸态（Z）。调试时可通过 /proc 观察到该状态。为避免遗留，父进程在终止流水线时执行“先 SIGTERM，超时后 SIGKILL，最后 wait 回收”的策略，并在控制面展示退出码与结束耗时，形成可验证的闭环 [7, 1].

5.4 运行结果与现象分析

本章结合实际运行现象说明三种通信方式的差异与共性。

5.4.1 样例与配置设计

为便于给出“可复现且差异明显”的实验结果，本节采用“更换输入 + 更换流水线配置”的方式构造对照组。由于各 Stage 的接口统一为字节流输入输出，不同输入与不同 Stage 链会在词频统计结果上体现出可解释的差异，从而支撑第 4 章的设计与实现分析。表 10 给出实验分组与预期差异点。

表 10　实验分组：输入与流水线配置对照

实验组	输入特征	Stage 链（按执行顺序）
E1（基准）	大小写混合、含标点与重复词	source.file → tokenize.en_words → count.tokens → report.text
E2（长度过滤）	同 E1	source.file → tokenize.en_words → token.len_range → count.tokens → report.text
E3（黑名单过滤）	含高频无信息词（如 the/of/and）与业务词混杂	source.file → tokenize.en_words → token.drop_list → count.tokens → report.text
E4（通信切换）	固定输入与固定 Stage 链（推荐用 E2）	run.mode 取 pipe/direct/indirect 对比输出一致性

5.4.2 三种通信方式的对照观察点

三种通信方式在业务结果上应保持一致，但在“进程数量、连接拓扑、可观测现象”上存在差异。表 11 给出对照观察点，用于本章实验分析。

表 11　三种通信方式对照观察点

通信方式	额外进程	观察现象
pipe	无	关注 EOF 传播与管道缓冲；父进程读取最终 stdout 以避免阻塞。
direct	无	关注 socketpair 点对点连接；与 pipe 相比拓扑相似但语义更通用。
indirect	mailboxd	关注 mailboxd 的出现与退出；关注 pass_fds 与 FD 关闭对 EOF 的影响。

5.4.3 实验：背压阻塞复现：pipe/direct

在 pipe 或 direct 模式下，将 debug.delay 放在流水线中后部（例如靠近报告生成之前），可观察到上游阶段出现明显的“写入受阻”。该现象对应内核在管道/套接字缓冲上的背压机制：缓冲区满则写端阻塞，直到读端释放空间 [8]。为减少截图数量，本实验不单独放图。若需要证据，可补充一张“阶段停止推进但 /proc 状态为 S”的截图，用于对应背压机制的说明。

5.4.4 实验：EOF 传播问题的定位：故障注入

将某个下游 Stage 强制终止（例如 SIGKILL），然后观察上游与父进程的行为：若父进程仍持有写端 FD 引用，则其他下游进程可能迟迟收不到 EOF 而持续等待输入。通过“父进程关闭多余 FD + wait 回收”的策略，可使流水线在异常路径上稳定收敛并给出明确退出状态。为减少截图数量，本实验以文字给出结论：关闭父进程中不再需要的写端引用后，下游进程可按预期收到 EOF 并结束；若未关闭，则下游可能持续等待输入。

5.4.5 Web 控制面：配置摘要与运行面板

本文实验展示统一采用 Web 控制面截图。

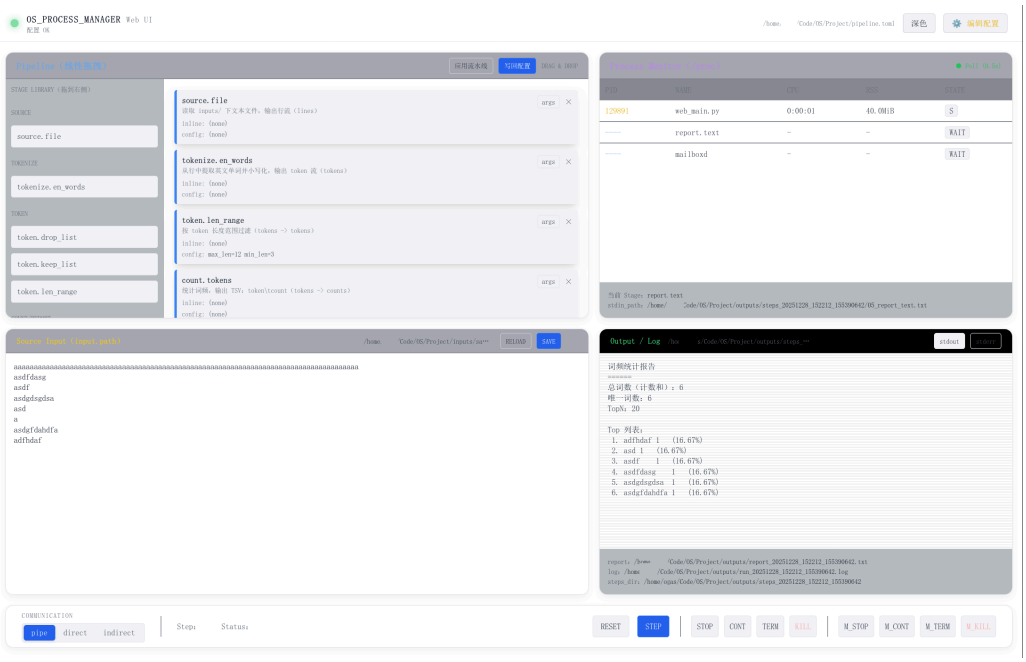

OS_PROCESS_MANAGER Web UI
source.file
tokenize.en_words
token.len_range
count.tokens
token.drop_list
token.keep_list
token.len_range
aaaa



图 8　单步执行：/proc 进程信息展示与运行中控制

与“单步执行：进程信息与输出预览”的截图合并：在同一张截图中体现“信号按钮 + /proc 状态字段”的变化即可。

5.4.8　实验：TERM/KILL 终止与回收

终止实验分两步：先发送 SIGTERM，观察 Stage 是否优雅退出；若在短时间内未退出，则发送 SIGKILL 强制终止。随后父进程通过 wait 回收退出状态，确保不遗留僵尸进程。该实验对应“终止与回收”的 OS 原理点 [7, 1]. 为控制篇幅，本实验不单独放图；在你补充截图时，可在同一张控制面截图中同时呈现“终止按钮、退出码、/proc 状态消失”三个要素。

5.4.9　三种通信方式的可切换性

通信方式由配置文件字段控制。实验中可将 run.mode 分别设置为 pipe/direct/indirect，在相同输入与相同 Stage 链的情况下得到一致的业务输出，从而证明“通信机制可运行时选择且不影响业务逻辑”。

5.4.10　实验结果截图

图 9 给出实验结果截图。E1–E3 重点体现“输入与流水线配置变化导致输出差异”；E4 体现“通信方式切换不改变业务输出”。

截图占位：assets/webui_exp_e1_baseline.png

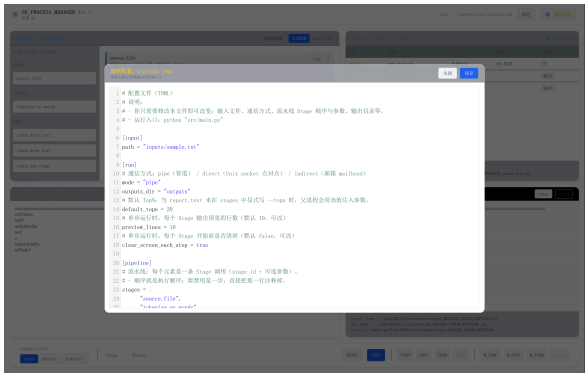
(a) E1：基准流水线输出结果

截图占位：assets/webui_exp_e2_len_range.png

(b) E2：长度过滤后的输出差异

截图占位：assets/webui_exp_e3_drop_list.png

(c) E3：黑名单过滤后的输出差异



(d) E4：运行时切换通信方式

图 9　实验结果截图

第六章　课设小结

本课程设计完成了题目要求的“进程创建、控制、终止和通信”核心目标：父进程能够创建与管理多个子进程，并在运行时切换三种通信方式完成同一流水线任务；系统支持进程暂停/继续/终止等控制，能基于 /proc 展示关键进程信息，并将运行现象与进程机制对应。

6.1　完成情况与效果

从功能点角度看，本设计完成了进程创建、控制、终止与回收、三种通信方式运行时切换以及基于 /proc 的进程信息展示；在相同输入与相同 Stage 链下，不同通信方式能够得到一致的业务输出，从而形成可验证的闭环。

6.2　创新点与加分项

在满足题目要求的基础上，本设计采用配置驱动与单步执行机制：运行参数统一由 TOML 配置管理，单步模式可在运行中发送信号并即时观察 /proc 状态与资源统计变化，从而增强可观测性。

6.3　理论对应与个人收获

通过本次课程设计，我对操作系统中“进程作为资源容器”的理解更具体化，尤其体现在以下几个可观测点：

1. fork/exec/wait 的闭环：创建子进程只是开始，等待并回收退出状态是生命周期管理的关键，否则会产生僵尸进程；
2. FD 抽象的一致性：pipe、socket、普通文件在用户态都表现为 FD 读写，从而可以用统一的字节流接口抽象三种通信方式；
3. EOF 与背压机制：EOF 是否传播取决于写端是否全部关闭；缓冲区满导致阻塞是内核层面的背压，而非业务逻辑卡死；
4. 信号与状态机：STOP/CONT/TERM/KILL 的语义差异可映射到 /proc 状态与退出码，使控制与现象之间形成对应关系。

不足之处在于：受限于课程设计时间与“稳定复现”的目标取向，部分模块仍以“可读性优先”的方式实现，尚可在健壮性与可扩展性方面继续改进，例如更精细的错误分类与恢复策略、更丰富的性能观测指标、以及更完善的自动化测试与实验对照数据。

6.4 改进方向

后续可从以下方面进一步完善：

1. 更完善的子子通信验证：在保持接口一致的前提下，加入独立的“子进程 ↔ 子进程”收发闭环；
2. 更丰富的观测指标：展示更多 /proc 字段或结合 ps 输出对照，提高可解释性；
3. 更系统的实验数据：对比三种通信方式的性能与资源占用（如吞吐、延迟、CPU 时间、RSS），形成数据化结论。

参考文献

[1] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating System Concepts. Wiley, 10 edition, 2018.

[2] Andrew S. Tanenbaum and Herbert Bos. Modern Operating Systems. Pearson, 4 edition, 2015.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Operating Systems: Three Easy Pieces. Arpaci-Dusseau Books, 2018.

[4] Randal E. Bryant and David R. O'Hallaron. Computer Systems: A Programmer's Perspective. Pearson, 3 edition, 2015.

[5] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. Communications of the ACM, 17(7):365–375, 1974.

[6] Maurice J. Bach. The Design of the UNIX Operating System. Prentice Hall, 1986.

[7] W. Richard Stevens and Stephen A. Rago. Advanced Programming in the UNIX Environment. Addison-Wesley Professional, 3 edition, 2013.

[8] Michael Kerrisk. The Linux Programming Interface: A Linux and UNIX System Programming Handbook. No Starch Press, 2010.

[9] IEEE and The Open Group. IEEE Std 1003.1-2017 (POSIX.1): Standard for Information Technology—Portable Operating System Interface (POSIX), 2017.

[10] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. UNIX Network Programming, Volume 1: The Sockets Networking API. Addison-Wesley Professional, 3 edition, 2003.

[11] Daniel P. Bovet and Marco Cesati. Understanding the Linux Kernel. O'Reilly Media, 3 edition, 2005.

[12] Robert Love. Linux Kernel Development. Addison-Wesley Professional, 3 edition, 2010.

附录

程序清单

本附录收录核心模块与典型 Stage 的主要源代码或核心节选，文件列表如下：

1. src/main.py
2. src/pipeline_runner.py
3. src/mailboxd.py
4. src/procfs.py
5. src/stage_count_tokens.py
6. src/stage_debug_delay.py

1. src/main.py

```
#!/usr/bin/env python3
# 用法：
#   python ”src/main.py”
#   python ”src/main.py” --config ”pipeline.toml”
#
# 说明：
#   - 父进程主程序入口：
#       * 提供交互菜单
#       * 创建/控制/终止 Stage 子进程
#       * 运行时可选三种通信方式（pipe/direct/indirect）
#       * 所有可配置项统一从配置文件读取（默认：pipeline.toml）

from __future__ import annotations

import argparse
from pathlib import Path
import sys

from ui_cli import run_cli


def parse_args(argv: list[str]) -> argparse.Namespace:
    parser = argparse.ArgumentParser(prog=”main”)
    parser.add_argument(
        ”--config”,
        default=”pipeline.toml”,
        help=”TOML 配置文件路径（默认：pipeline.toml）”,
    )
    return parser.parse_args(argv)


def main() -> int:
    args = parse_args(sys.argv[1:])
    return run_cli(config_path=Path(args.config))


if __name__ == ”__main__”:
    raise SystemExit(main())
```

2. src/pipeline_runner.py

```
# 说明：
#  - 本节给出 pipeline_runner.py 的更精简节选，重点聚焦：
#      * 三种通信方式的“构造思路”（pipe / direct / indirect）
#      * 终止策略：先 SIGTERM 并等待（超时），再 SIGKILL，最后 wait 回收
#  - 为控制篇幅，省略预览/落盘、异常清理等工程性细节。

from __future__ import annotations

import os
import signal
import socket
import subprocess
import time
from dataclasses import dataclass
from pathlib import Path
from typing import Optional


MODE_PIPE = "pipe"
MODE_DIRECT = "direct"
MODE_INDIRECT = "indirect"


@dataclass
class PipelineRun:
    mode: str
    stages: list[subprocess.Popen]
    mailbox: Optional[subprocess.Popen]

    def send_signal_all(self, sig: int) -> None:
        for p in self.stages:
            if p.poll() is None:
                try:
                    os.kill(p.pid, sig)
                except ProcessLookupError:
                    pass
        if self.mailbox is not None and self.mailbox.poll() is None:
            try:
                os.kill(self.mailbox.pid, sig)
            except ProcessLookupError:
                pass

    def terminate_all(self, timeout_seconds: float = 2.0) -> None:
        self.send_signal_all(signal.SIGTERM)

        deadline = time.time() + max(0.0, float(timeout_seconds))
        while time.time() < deadline:
            stages_exited = all(p.poll() is not None for p in self.stages)
            mailbox_exited = self.mailbox is None or self.mailbox.poll() is not None
            if stages_exited and mailbox_exited:
                break
            time.sleep(0.05)

        self.send_signal_all(signal.SIGKILL)

        for p in self.stages:
            try:
                p.wait(timeout=1)
            except Exception:
                pass
        if self.mailbox is not None:
            try:
                self.mailbox.wait(timeout=1)
            except Exception:
                pass


def start_pipeline(
    mode: str,
    stage_argvs: list[list[str]],
    input_path: Path,
    log_path: Path,
) -> PipelineRun:
    log_f = open(log_path, "ab", buffering=0)
    stages: list[subprocess.Popen] = []
    mailbox: Optional[subprocess.Popen] = None

    if mode == MODE_PIPE:
        # pipe：前一 Stage 的 stdout 作为下一 Stage 的 stdin（Unix pipeline）
        prev = open(input_path, "rb")
        for argv in stage_argvs:
            p = subprocess.Popen(
```

```
            argv,
            stdin=prev,
            stdout=subprocess.PIPE,
            stderr=log_f,
            close_fds=True,
        )
        stages.append(p)
        prev = p.stdout

    elif mode == MODE_DIRECT:
        # direct：socketpair 点对点直连（本设计仍按“单向数据流”使用）
        a, b = socket.socketpair()
        p1 = subprocess.Popen(
            stage_argvs[0],
            stdin=open(input_path, ”rb”),
            stdout=a,
            stderr=log_f,
            close_fds=True,
            pass_fds=(a.fileno(),),
        )
        stages.append(p1)
        a.close()
        # 下游 Stage 以 b 作为 stdin（示意，省略剩余 Stage 的循环拼装）
        b.close()

    elif mode == MODE_INDIRECT:
        # indirect：插入 mailboxd；父进程 pass_fds 把路由所需 FD 显式传入
        mailbox = subprocess.Popen(
            [”python”, ”src/mailboxd.py”, ”--route”, ”3:4”],
            stdin=subprocess.DEVNULL,
            stdout=subprocess.DEVNULL,
            stderr=log_f,
            close_fds=True,
            pass_fds=(3, 4),
        )
        # 省略：父进程再启动各 Stage，并把其 stdin/stdout 重定向到对应 FD

    else:
        raise ValueError(f”unsupported mode: {mode}”)

    return PipelineRun(mode=mode, stages=stages, mailbox=mailbox)
```

(3) src/mailboxd.py（节选：路由转发核心）

```
#!/usr/bin/env python3
# 说明：
#   - mailboxd 是“间接通信”的中介进程：按 route(in_fd:out_fd) 转发字节流。
#   - 为控制篇幅，以下仅保留核心循环：read -> write，直到 EOF/退出。

from __future__ import annotations

import os
import sys
import threading
from dataclasses import dataclass


@dataclass(frozen=True)
class Route:
    in_fd: int
    out_fd: int


def forward_stream(route: Route, stop_event: threading.Event) -> None:
    try:
        while not stop_event.is_set():
            chunk = os.read(route.in_fd, 64 * 1024)
            if not chunk:
                break
            view = memoryview(chunk)
            while view and not stop_event.is_set():
                n = os.write(route.out_fd, view)
                view = view[n:]
    finally:
        for fd in (route.in_fd, route.out_fd):
            try:
                os.close(fd)
```

```
            except OSError:
                pass


def main(argv: list[str]) -> int:
    # argv 示例：["--route","3:4","--route","5:6",...]
    routes: list[Route] = []
    it = iter(argv)
    for x in it:
        if x == "--route":
            in_str, out_str = next(it, "").split(":", 1)
            routes.append(Route(int(in_str), int(out_str)))

    stop_event = threading.Event()
    threads = [threading.Thread(target=forward_stream, args=(r, stop_event), daemon=True) for r in routes]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    return 0


if __name__ == "__main__":
    raise SystemExit(main(sys.argv[1:]))
```

(4) src/procfs.py（节选：read_proc_info）

```
# 说明：
#   - read_proc_info 读取 /proc/<pid>/status 与 /proc/<pid>/cmdline 的关键字段，
#     用于展示：PID/PPID/状态/内存 RSS/命令行。

from __future__ import annotations

from dataclasses import dataclass
from typing import Optional


@dataclass(frozen=True)
class ProcInfo:
    pid: int
    ppid: int
    state: str
    cmdline: str
    rss_bytes: Optional[int]


def _read_text(path: str) -> str:
    with open(path, "r", encoding="utf-8", errors="replace") as f:
        return f.read()


def _read_bytes(path: str) -> bytes:
    with open(path, "rb") as f:
        return f.read()


def read_proc_info(pid: int) -> Optional[ProcInfo]:
    proc_dir = f"/proc/{pid}"
    try:
        status = _read_text(f"{proc_dir}/status")
    except OSError:
        return None

    state = "?"
    ppid = 0
    rss_bytes: Optional[int] = None
    for line in status.splitlines():
        if line.startswith("State:"):
            state = line.split()[1]
        elif line.startswith("PPid:"):
            ppid = int(line.split()[1])
        elif line.startswith("VmRSS:"):
            rss_bytes = int(line.split()[1]) * 1024

    cmdline = ""
    try:
        raw = _read_bytes(f"{proc_dir}/cmdline")
        parts = [p.decode("utf-8", errors="replace") for p in raw.split(b"\0") if p]
```

```
        cmdline = " ".join(parts)
    except OSError:
        pass

    return ProcInfo(pid=pid, ppid=ppid, state=state, cmdline=cmdline, rss_bytes=rss_bytes)
```

(5) src/stage_count_tokens.py

```
#!/usr/bin/env python3
# 用法：
#   python "src/stage_count_tokens.py" < tokens.txt > counts.tsv
#
# 说明：
#   - Stage：count.tokens
#   - 输入：tokens（stdin，每行一个 token）
#   - 输出：counts（stdout，TSV：token\\tcount）
#   - 对应 Stream API：
#       * collect(groupingBy(identity, counting))

from __future__ import annotations

import sys
from collections import Counter


def main() -> int:
    counter: Counter[str] = Counter()
    for line in sys.stdin:
        token = line.strip()
        if not token:
            continue
        counter[token] += 1

    # 输出稳定：按 token 字母序输出，便于 diff / 调试。
    for token in sorted(counter.keys()):
        sys.stdout.write(f"{token}\t{counter[token]}\n")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

(6) src/stage_debug_delay.py

```
#!/usr/bin/env python3
# 用法：
#   python "src/stage_debug_delay.py" --seconds 0.1 < in.txt > out.txt
#
# 说明：
#   - Stage：debug.delay（实验辅助）
#   - 输入：任意（stdin）
#   - 输出：透传（stdout）
#   - 目的：
#       * 让流水线运行更久，便于观察 stop/cont/term/kill 控制效果

from __future__ import annotations

import argparse
import sys
import time


def parse_args(argv: list[str]) -> argparse.Namespace:
    parser = argparse.ArgumentParser(prog="stage_debug_delay")
    parser.add_argument(
        "--seconds",
        type=float,
```

```
        default=0.1,
        help=" 每行延时秒数（默认 0.1）",
    )
    return parser.parse_args(argv)


def main(argv: list[str]) -> int:
    args = parse_args(argv)
    if args.seconds < 0:
        print(f"[debug.delay] seconds 不能为负数：{args.seconds}", file=sys.stderr)
        return 2

    for line in sys.stdin:
        sys.stdout.write(line)
        sys.stdout.flush()
        time.sleep(args.seconds)
    return 0


if __name__ == "__main__":
    raise SystemExit(main(sys.argv[1:]))
```