苏州大学实验报告

院、系	计算机学院	年级专业	19 计科图灵	姓名	张昊	学号	1927405160
课程名称	操作系统课程实践					成绩	
指导教师	李培峰	同组实验者	无		实验日期	2022 年 3 月 10 日	

实 验 名 称　　实验 2　Linux 进程创建和和调度模拟

一. 实验目的

1. 加深对进程概念的理解，进一步认识并发执行的实质，明确进程和程序的区别。
2. 掌握 Linux 操作系统中进程的创建和终止操作。
3. 掌握在 Linux 操作系统中创建子进程并加载新映像的操作。
4. 深入理解系统如何组织进程。
5. 理解常用进程调度算法的具体实现。

二. 实验内容

1. （**实验 3.1: 进程的创建**）
   (1) 编写一个 C 程序，并使用系统调用 `fork()` 创建一个子进程。要求如下:
      a) 在子进程中分别输出当前进程为子进程的提示、当前进程的 PID 和父进程的 PID、根据用户输入确定当前进程的返回值、退出提示等信息。
      b) 在父进程中分别输出当前进程为父进程的提示、当前进程的 PID 和子进程的 PID、等待子进程退出后获得的返回值、退出提示等信息。
   (2) 编写另一个 C 程序，使用系统调用 `fork()` 以创建一个子进程，并使用这个子进程调用 exec 函数族以执行系统命令 ls。
2. （**实验 3.2: 进程调度算法的模拟**）
   编写 C 程序，模拟实现单处理器系统中的进程调度算法，实现对多个进程的模拟调度，要求采用常见的进程调度算法(如先来先服务、时间片轮转和优先级等调度算法)进行模拟调度。

三. 操作方法和实验步骤

1. （**实验 3.1: 进程的创建**）
   本实验的主要目的是验证系统调用 `fork()` 的返回值。
   (1) 在主程序中通过 `fork()` 创建子进程，并根据 `fork()` 的返回值确定所处的进程是子进程还是父进程；分别在子进程和父进程中调用 `getpid()`、`getppid()`、`wait()` 等函数进行验证。实验代码如下：

```
/* fork/process-fork.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  pid_t child_pid = fork();  // 创建子进程，获取 PID
```

```
  if (child_pid == 0) { // 当前处在子进程中
    int ret;  // 子进程返回值
    printf("[CHILD] 当前处于子进程中\n");
    printf("[CHILD] 子进程 PID = %d\n", getpid());
    printf("[CHILD] 父进程 PID = %d\n", getppid());
    printf("[CHILD] 子进程睡眠 1 秒...\n");
    sleep(1);
    printf("[CHILD] 请输入子进程执行完毕后的返回值(0-255): ");
    scanf("%d", &ret);
    printf("[CHILD] 子进程即将退出\n");
    exit(ret);  // 退出子进程
  } else if (child_pid > 0) { // 当前处在父进程中, fork()返回子进程 PID
    int status;  // 子进程向父进程提供的退出状态
    printf("[PARENT] 当前处于父进程中\n");
    printf("[PARENT] 父进程 PID = %d\n", getpid());
    printf("[PARENT] 子进程 PID = %d\n", child_pid);
    printf("[PARENT] 父进程将等待子进程运行结束...\n");
    wait(&status);
    printf("[PARENT] 子进程的返回值: %d\n", WEXITSTATUS(status));
    printf("[PARENT] 父进程即将退出\n");
    exit(EXIT_SUCCESS);
  } else {  // fork 失败
    fprintf(stderr, "fork 失败! \n");
    exit(EXIT_FAILURE);
  }
}
```

(2) 在主程序中通过 fork() 创建子进程；在子进程中使用 exec 函数族中 execvp 函数，通过传入以 NULL 结尾的数组 argv 来执行命令 ls。若可执行文件存在将转向进程代码，并在运行结束后退出；否则返回-1。实验代码如下：

```
/* fork/process-exec.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  pid_t child_pid = fork();  // 创建子进程，获取 PID
  if (child_pid == 0) { // 当前处在子进程中
    printf("[CHILD] 当前处于子进程中\n");
    // ls 命令所需的参数，最后一个元素必须是 NULL
    char *argv[] = {"ls", "-l", "-a", "-h", NULL};
    if (-1 == execvp("/usr/bin/ls", argv)) {  // 执行命令 ls
      // 执行不成功会进入分支，否则子进程会 exit
      fprintf(stderr, "[CHILD] 子进程执行失败! \n");
```

```
    exit(EXIT_FAILURE);
  }
 } else if (child_pid > 0) { // 当前处在父进程中, fork()返回子进程 PID
  int status;  // 子进程向父进程提供的退出状态
  printf("[PARENT] 当前处于父进程中\n");
  printf("[PARENT] 父进程将等待子进程运行结束...\n");
  wait(&status);
  printf("[PARENT] 子进程的返回值: %d\n", WEXITSTATUS(status));
  printf("[PARENT] 父进程即将退出\n");
  exit(0);
 } else {  // fork 失败
  fprintf(stderr, "fork 失败! \n");
  exit(EXIT_FAILURE);
 }
}
```

2. （**实验 3.2: 进程调度算法的模拟**）

首先是进程控制块 PCB 的设计，根据调度算法，最后包含的信息有 PID、名称、进程状态、优先数、到达时间、剩余区间时间、结束时间等，使用 C 语言结构体来实现：

```
/* schedule/schedule_proc.h */
// 进程 PCB
typedef struct pcb_t {
 unsigned pid; // PID
 char name[16];  // 进程名称
 states_t states; // 进程的当前状态
 int priority; // 进程优先数
 long arrive_time; // 到达时间
 long interval_time; // 剩余区间时间
 long finish_time;  // 结束时间，-1 为尚未结束运行
 struct pcb_t *next_proc; // 所有进程(以 PCB 的形式)组成的链表
} pcb;
```

对于进程的状态，定义了枚举类型来表示进程的新建、就绪、运行、等待和完成：

```
/* schedule/schedule_proc.h */
// 进程运行状态
typedef enum states {
 PROC_STATE_CREATED, PROC_STATE_READY, PROC_STATE_RUNNING,
 PROC_STATE_WAIT, PROC_STATE_FINISH
} states_t;
```

其次是就绪队列 ready 的设计，使用单向链表来表示。在 C 语言中使用指向队列首部进程的指针和 PCB 中指向下一个进程的指针来实现；为了实现 FIFO 队列，还设计了指向队列尾部元素的指针 ready_tail。

假定在各调度算法中，对于每个需要调度的都会提供进程名、到达时间、区间时间这三个信息。因此设计了一个新建队列 created，用于保存用户输入的但是还未到到达时间到进程，以及使用一个全局变量 machine_time 保存处理机时间（整个程序已经运行了多长 CPU 时间）。

在实现具体的调度算法之前，定义了一些在调度算法中会使用到的函数。

进程级别的函数有创建进程、回收进程资源、显示进程信息和运行空闲进程，其中运行空闲

进程函数是在就绪队列空闲，但新建队列中下一个进程还没有到到达时间时运行的函数。各函数的实现如下：

```
/* schedule/schedule_proc.c */
/**
 * @brief 创建进程
 * @param pid PID
 * @param name 进程名称
 * @param priority 优先级
 * @param arrive 到达时间
 * @param interval 区间时间
 * @return 若创建成功，返回进程 PCB；否则为 NULL
 */
pcb *create_proc(unsigned pid, const char *name, int priority,
                 long arrive, long interval) {
  pcb *proc = (pcb *) malloc(sizeof(pcb));
  if (NULL != proc) {
    proc->states = PROC_STATE_CREATED;
    proc->pid = pid;
    strcpy(proc->name, name);
    proc->priority = priority;
    proc->arrive_time = arrive;
    proc->interval_time = interval;
    proc->finish_time = -1;
    proc->next_proc = NULL;
  }
  return proc;
}
/**
 * @brief 回收进程资源
 * @param proc 进程 PCB
 * @return 若 PCB 不为 NULL，返回链表后一个进程 PCB 指针；否则为 NULL
 */
pcb *destroy_proc(pcb *proc) {
  pcb *next_proc = NULL;
  if (NULL != proc) {
    next_proc = proc->next_proc;
    free(proc);
  }
  return next_proc;
}
/**
 * @brief 显示进程信息
 * @param proc 进程 PCB
 */
void display_proc(pcb *proc) {
```

```
  if (NULL != proc) {
    printf("PID: %-d\t PROCESS: %-s\t PRIOR: %-d\t ",
           proc->pid, proc->name, proc->priority);
    char *states_string;
    switch (proc->states) {
      case PROC_STATE_CREATED:  states_string = "CREATED";  break;
      case PROC_STATE_READY:  states_string = "READY";  break;
      case PROC_STATE_RUNNING:  states_string = "RUNNING";  break;
      case PROC_STATE_WAIT:  states_string = "WAIT";  break;
      case PROC_STATE_FINISH:  states_string = "FINISH";  break;
    }
    printf("STATES: %-8s\t Arrive: %-ld\t Interval: %-ld\t "
           "Finish: %-ld\n", states_string, proc->arrive_time,
           proc->interval_time, proc->finish_time);
  }
}
/**
 * @brief 运行空闲进程
 * @param interval 持续时长
 */
void idle_proc(long interval) {
  // running IDLE process ${interval} second(s)
  machine_time += interval;
  printf("CPU 空闲时长: %ld\n", interval);
}
```

队列级别的函数有入队、出队（均以 FIFO 方式），调整就绪队列队尾指针，对进程链表排序，将当前时间点以及之前到达的进程加入到就绪队列以及显示就绪队列等函数。排序算法选用的是归并排序，原因是对于链表的排序，归并排序可以在$O(n\log n)$的时间和$O(1)$的空间内完成对进程 PCB 的稳定排序，其中比较函数由第二个参数给出。应该注意的是，若对就绪队列排序，需要重新调整队尾指针。各函数的实现如下：

```
/* schedule/schedule_proc.c */
/**
 * @brief 就绪队列入队
 * @param proc 进程 PCB
 */
void push_ready(pcb *proc) {
  if (NULL == proc || 0 == proc->interval_time) {
    return;  // 空进程和区间为 0 的进程不会加入就绪队列
  }
  proc->next_proc = NULL;
  proc->states = PROC_STATE_READY;  // 设置状态为就绪
  if (NULL == ready) {  // 就绪队列为空，同时设置队尾指针
    ready = proc;
    ready_tail = proc;
  } else {  // 向队尾指针后追加，并调整队尾指针
```

```
    ready_tail->next_proc = proc;
    ready_tail = ready_tail->next_proc;
  }
}
/**
 * @brief 就绪队列出队
 * @return 若就绪队列为空返回 NULL；否则返回第一个进程
 */
pcb *pop_ready() {
  if (ready == NULL) {
    return NULL;
  }
  pcb *proc = ready;
  ready = ready->next_proc;
  proc->next_proc = NULL;
  if (proc == ready_tail) {  // 若就绪队列只有一个进程（队首队尾为同一个进程）
    ready_tail = NULL;  // 出队时队尾指针置空
  }
  return proc;
}
/**
 * @brief 【排序辅助函数】找中点，拆分链表
 * @param head 进程链表
 * @return 第二部分的链表首元素
 */
pcb *sort_list_search_mid(pcb *head) {
  pcb *fast = head, *slow = head;
  pcb *mid = NULL, *mid_after = NULL;
  while (NULL != fast && NULL != fast->next_proc
      && NULL != fast->next_proc->next_proc) {
    fast = fast->next_proc->next_proc;
    slow = slow->next_proc;
  }
  mid = slow;
  mid_after = mid->next_proc;
  mid->next_proc = NULL; // 将链表一分为二
  return mid_after;
}
/**
 * @brief 【排序辅助函数】合并两个有序链表
 * @param left 有序链表 1
 * @param right 有序链表 2
 * @param comparable 比较两 PCB 的回调函数
 * @return 合并后的头指针
 */
```

```
pcb *sort_list_merge(pcb *left, pcb *right,
                     pcb_comparable_t comparable) {
  pcb *p1 = left, *p2 = right;
  pcb result, *p3 = &result;  // 不包含数据的头节点
  // p3 在合并过程中，始终指向合并链表的当前尾节点
  while (NULL != p1 && NULL != p2) {
    if (comparable(p1, p2) <= 0) {
      p3->next_proc = p1;
      p3 = p1;
      p1 = p1->next_proc;
    } else {
      p3->next_proc = p2;
      p3 = p2;
      p2 = p2->next_proc;
    }
  }
  if (NULL == p2) { p3->next_proc = p1; }
  if (NULL == p1) { p3->next_proc = p2; }
  left = result.next_proc;
  return left;
}
/**
 * @brief 对进程队列排序（归并排序）
 * @note 排序后需要调整链表的尾指针（如果有的话）
 * @param head 进程链表
 * @param comparable 比较两 PCB 的回调函数
 * @return 若进程链表不为 NULL，返回排序后的链表首指针；否则为 NULL
 */
pcb *sort_list(pcb *head, pcb_comparable_t comparable) {
  if (NULL == head || NULL == head->next_proc) {
    return head;  // 递归终止条件：链表长度小于等于 1
  }
  pcb *left, *right;
  left = head;
  right = sort_list_search_mid(head);
  left = sort_list(left, comparable);
  right = sort_list(right, comparable);
  head = sort_list_merge(left, right, comparable);
  return head;
}
/**
 * @brief 重置就绪队列队尾指针
 */
void reset_ready_tail() {
  if (NULL == ready) {
```

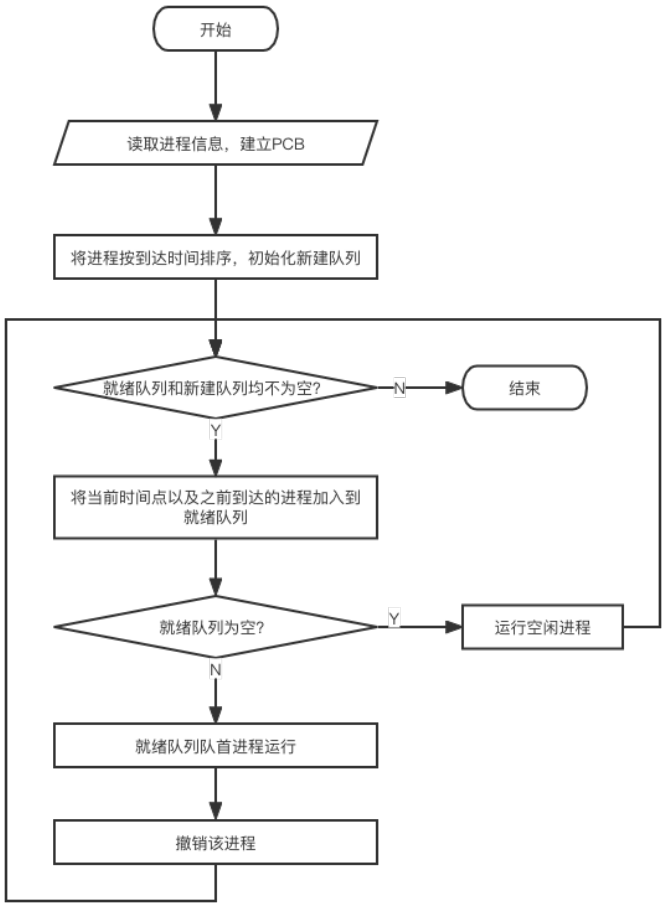
```
    ready_tail = NULL;
    return;
  }
  pcb *proc = ready;
  while (NULL != proc) {
    ready_tail = proc;
    proc = proc->next_proc;
  }
}
/**
 * @brief 将当前时间点以及之前到达的 PCB 加入到就绪队列
 */
void new_proc_ready() {
  pcb *p;
  while (NULL != created && created->arrive_time <= machine_time) {
    p = created;
    created = created->next_proc;
    push_ready(p);
  }
}
/**
 * @brief 显示就绪队列
 */
void display_ready() {
  if (NULL != ready) {
    printf("\n*** 就绪队列 (在处理机时间%ld) ***\n", machine_time);
  } else {
    printf("\n*** 就绪队列为空 (在处理机时间%ld) ***\n", machine_time);
  }
  pcb *p = ready;
  int count = 0;
  while (NULL != p) {
    display_proc(p);
    p = p->next_proc;
    count++;
  }
  printf("*** 就绪队列进程数量 %d ***\n\n", count);
}
```

在具体调度算法的实现中，各算法需定义进程读取函数、运行进程函数以及主函数。规定模拟程序从命令行参数中读取保存有进程信息的文件路径，通过读取文件内容建立进程。一般的进程信息的文件格式为：第一行为一个整数 N，表示文件中共有多少个进程，接下来 N 行为要调度的进程信息，一个进程占一行，格式由具体调度算法确定。所有建立的进程首先保存在新建队列中，在程序的运行过程中不断将当前时间点以及之前到达的进程加入到就绪队列（即只有进程的到达时间早于或等于处理机时间时才会被加入到就绪队列），并对按一定规则排序的就绪队列的队首进程进行调度，直至就绪队列和新建队列为空。若某一时刻无进程请求 CPU（即经过上

述调整后就绪队列为空而新建队列不为空），则处理机进入空闲。对于模拟程序的每一步调度，程序就输出一次运行进程和就绪队列中所有进程的信息。

本实验实现了先来先服务（FCFS）、时间片轮转（RR）、短作业优先（SJF）、最短剩余时间优先（SRTF）、动态优先级（PR）等调度算法，下面介绍各算法等具体运行情况。

(1) 先来先服务（FCFS）：按照进程请求 CPU 的先后顺序使用 CPU，进程在获得 CPU 后将一直运行至结束，最后撤销该进程。这是一种非抢占式的调度。使用 FCFS 算法的调度程序的流程图如下：



规定进程信息文件中描述进程的格式（使用空格分割）：

```
进程名称 到达时间 区间时间
```

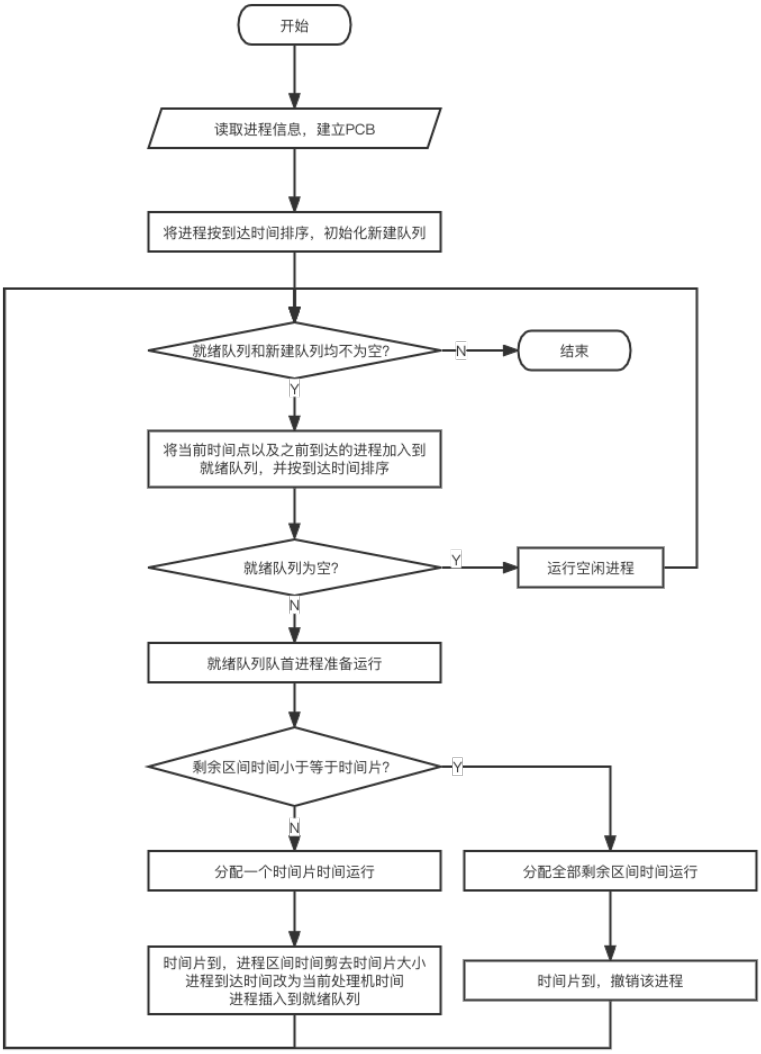
代码如下：

```
/* schedule/schedule_fcfs.c */
#include <stdio.h>
#include <stdlib.h>
#include "schedule_proc.h"
/** 按到达时间比较进程 PCB */
long compare_early_arrive(pcb *p1, pcb *p2) {
  return p1->arrive_time - p2->arrive_time;
}
/** 进程读取函数 */
int initialize(const char *filename) {
  FILE *fp = fopen(filename, "r");
  if (NULL == fp) {
```

```
    return EXIT_FAILURE;
  }
  int count;
  pcb *p = NULL;
  fscanf(fp, "%d", &count);
  for (unsigned pid = 1; pid <= count; ++pid) {
    char name[16]; long arrive, interval;
    fscanf(fp, "%s %ld %ld", name, &arrive, &interval);
    if (NULL == created) {  // 在新建队列等候
      created = create_proc(pid, name, 1, arrive, interval);
      p = created;
    } else {
      p->next_proc = create_proc(pid, name, 1, arrive, interval);
      p = p->next_proc;
    }
  }
  created=sort_list(created, compare_early_arrive); //按到达时间排序
  return EXIT_SUCCESS;
}
/** 运行进程 */
void run(pcb *proc) {
  if (NULL != proc) {
    proc->states = PROC_STATE_RUNNING;  // 设置进程运行状态
    printf("当前正在运行的进程是 %s: \n", proc->name); // 当前运行的进程
    display_proc(proc);
    machine_time += proc->interval_time;
    printf("进程 %s 运行时间: %ld\n",
           proc->name, proc->interval_time);
    proc->states = PROC_STATE_FINISH;  // 结束
    proc->finish_time = machine_time;
    printf("进程 %s 运行结束: \n", proc->name);
    display_proc(proc);
  }
}
int main(int argc, const char **argv) {
  if (argc != 2) {
    fprintf(stderr, "请提供一个命令行参数作为被调度进程信息! \n");
    return EXIT_FAILURE;
  }
  if (EXIT_SUCCESS != initialize(argv[1])) {
    fprintf(stderr, "文件读取失败! \n");
    return EXIT_FAILURE;
  }
  pcb *proc = NULL;  // 正在运行的进程
  while (NULL != ready || NULL != created) {  // 两队列均不为空才完成
```

```
    new_proc_ready();  // 将当前时间点以及之前到达的 PCB 加入到就绪队列
    // 无需再次排序，因为新建队列已经有序
    display_ready();
    if (NULL == ready) { // 若就绪队列为空，说明当前时间 CPU 空闲
      idle_proc(created->arrive_time - machine_time);
      continue;
    }
    proc = pop_ready();  // 就绪队列队首出队
    run(proc);  // 运行
    destroy_proc(proc);  // 回收进程资源
  }
  printf("\n*** 所有进程调度完成 ***\n");
  return EXIT_SUCCESS;
}
```

(2) 时间片轮转（RR）：按照进程请求 CPU 的先后顺序分配时间片，每个进程分配一定长度的 CPU 时间。若进程时间片结束仍未运行结束则抢占，并重新加入就绪队列；否则撤销进程。这是一种抢占式的调度。若有被抢占的进程和新进程同时要加入就绪队列，则被抢占的进程先加入就绪队列，同时被抢占的进程将改变到达时间为被剥夺 CPU 的时间，且总是被抢占的进程先加入就绪队列。使用 RR 算法的调度程序的流程图如下：



规定进程信息文件中描述进程的格式（使用空格分割）：

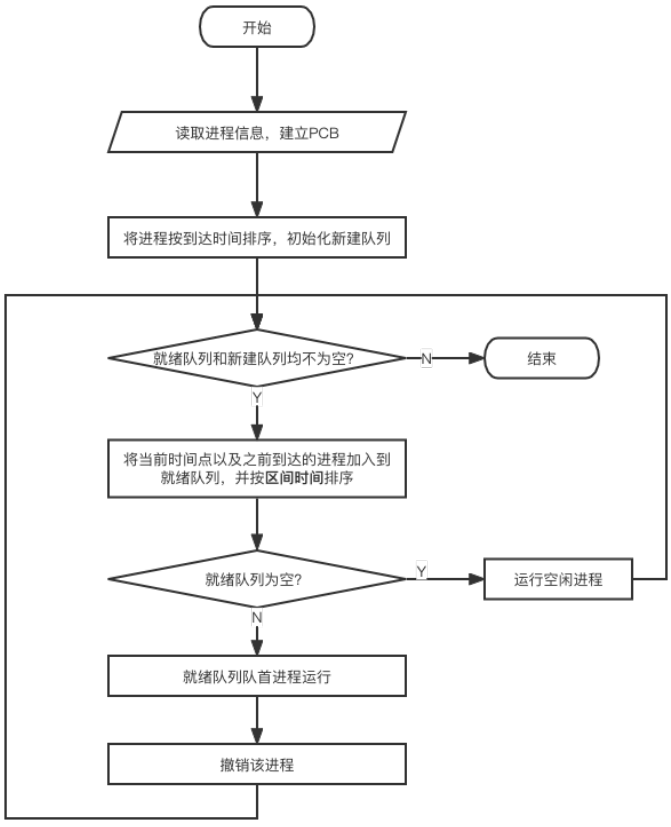
```
进程名称 到达时间 区间时间
```

代码如下：

```
/* schedule/schedule_rr.c */
#include <stdio.h>
#include <stdlib.h>
#include "schedule_proc.h"
#define SLICE 20  // 时间片大小
// 按到达时间比较进程 PCB 函数、进程读取函数与 FCFS 相同，在此省略
/**
 * @brief 以时间片轮转算法运行进程
 * @param proc 进程 PCB
 */
void run(pcb *proc) {
  if (NULL != proc) {
    proc->states = PROC_STATE_RUNNING;  // 设置进程运行状态
    printf("当前正在运行的进程是 %s: \n", proc->name); // 当前运行的进程
    display_proc(proc);
    if (proc->interval_time<=SLICE) {//剩余区间时间少于或等于一个时间片
      machine_time += proc->interval_time;  // 分配全部区间时间
      printf("进程 %s 运行时间: %ld\n",
             proc->name, proc->interval_time);
      proc->states = PROC_STATE_FINISH;  // 转为完成
      proc->finish_time = machine_time;
      printf("进程 %s 运行结束: \n", proc->name);
      display_proc(proc);
    } else {  // 只分配一个时间片
      machine_time += SLICE;
      proc->interval_time -= SLICE;
      printf("进程 %s 运行时间: %d\n", proc->name, SLICE);
      proc->arrive_time = machine_time;  // 修改到达时间
      push_ready(proc);  // 加入就绪队列
      printf("进程 %s 被抢占\n", proc->name);
    }
  }
}
int main(int argc, const char **argv) {
  if (argc != 2) {
    fprintf(stderr, "请提供一个命令行参数作为被调度进程信息！\n");
    return EXIT_FAILURE;
  }
  if (EXIT_SUCCESS != initialize(argv[1])) {
    fprintf(stderr, "文件读取失败！\n");
    return EXIT_FAILURE;
  }
```

```
  pcb *proc = NULL;
  while (NULL != ready || NULL != created) {//两队列均不为空才调度完成
    new_proc_ready();  // 将当前时间点以及之前到达的 PCB 加入到就绪队列
    // 按到达时间对就绪队列进行排序（因为新进程可能是在一个时间片内到达）
    ready = sort_list(ready, compare_early_arrive);
    reset_ready_tail();  display_ready();
    if (NULL == ready) { // 若就绪队列为空，说明当前时间 CPU 空闲
      idle_proc(created->arrive_time - machine_time);
      continue;
    }
    proc = pop_ready();  // 就绪队列队首出队
    run(proc);  // 运行一个时间片
    if (PROC_STATE_FINISH == proc->states) {  //运行结束，回收进程资源
      destroy_proc(proc);
    }
  }
  printf("\n*** 所有进程调度完成 ***\n");
  return EXIT_SUCCESS;
}
```

(3) 短作业优先（SJF）：每次调度下次运行 CPU 时间最短的进程，具体为对就绪队列按照区间时间进行排序，按照运行时间从小到大分配 CPU，进程在获得 CPU 后将一直运行至结束，最后撤销该进程。这是一种非抢占式的调度。使用 SJF 算法的调度程序的流程图如下：



规定进程信息文件中描述进程的格式（使用空格分割）：

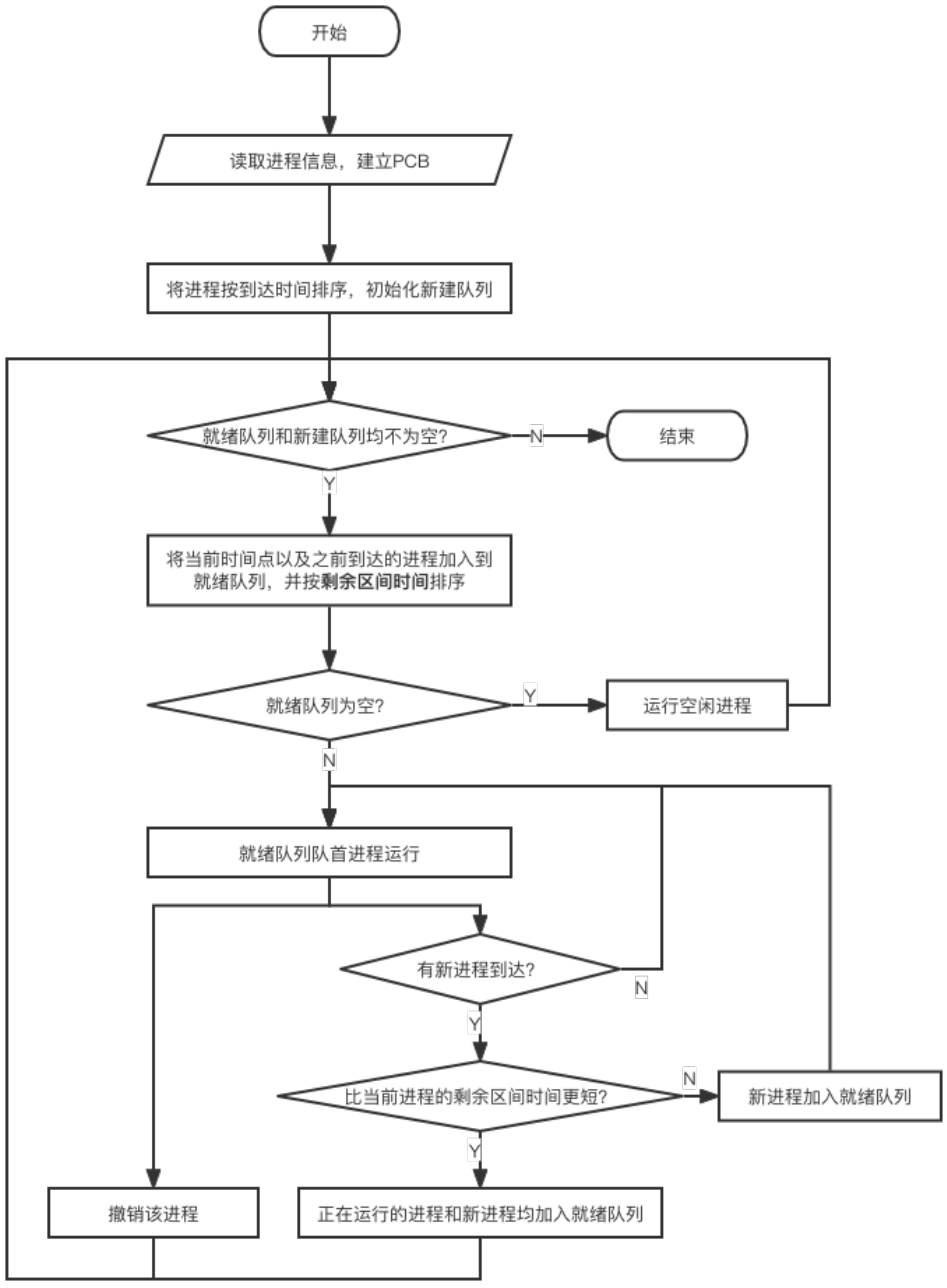
```
进程名称 到达时间 区间时间
```

代码如下：

```
/* schedule/schedule_sjf.c */
#include <stdio.h>
#include <stdlib.h>
#include "schedule_proc.h"
/** 按区间时间比较进程 PCB */
long compare_shortest_job(pcb *p1, pcb *p2) {
  return p1->interval_time - p2->interval_time;
}
// 按到达时间比较进程 PCB 函数、进程读取函数与 FCFS 相同，在此省略
/**
 * @brief 以 SJF 算法运行进程
 * @param proc 进程 PCB
 */
void run(pcb *proc) {
  if (NULL != proc) {
    proc->states = PROC_STATE_RUNNING;  // 设置进程运行状态
    printf("当前正在运行的进程是 %s: \n", proc->name); // 当前运行的进程
    display_proc(proc);
    machine_time += proc->interval_time;
    printf("进程 %s 运行时间: %ld\n",
           proc->name, proc->interval_time);
    proc->states = PROC_STATE_FINISH;  // 结束
    proc->finish_time = machine_time;
    printf("进程 %s 运行结束: \n", proc->name);
    display_proc(proc);
  }
}
int main(int argc, const char **argv) {
  if (argc != 2) {
    fprintf(stderr, "请提供一个命令行参数作为被调度进程信息! \n");
    return EXIT_FAILURE;
  }
  if (EXIT_SUCCESS != initialize(argv[1])) {
    fprintf(stderr, "文件读取失败! \n");
    return EXIT_FAILURE;
  }
  pcb *proc = NULL;
  while (NULL != ready || NULL != created) {//两队列均不为空才调度完成
    new_proc_ready();  // 将当前时间点以及之前到达的 PCB 加入到就绪队列
    // 按作业长短对就绪队列进行排序
    ready = sort_list(ready, compare_shortest_job);
    reset_ready_tail(); display_ready();
    if (NULL == ready) { // 若就绪队列为空，说明当前时间 CPU 空闲
      idle_proc(created->arrive_time - machine_time);
```

```
      continue;
    }
    proc = pop_ready();  // 就绪队列队首出队
    run(proc);  // 运行
    destroy_proc(proc);  // 回收进程资源
  }
  printf("\n*** 所有进程调度完成 ***\n");
  return EXIT_SUCCESS;
}
```

(4) 最短剩余时间优先（SRTF）是 SJF 调度算法的抢占式版本，整体思路与 SJF 相同，只是在比当前进程的剩余时间更短（严格小于）的进程到达时，调度新到达的进程运行。可以通过当有新进程到达时判断是否抢占，并将新进程加入就绪队列来实现。使用 SRTF 算法的调度程序的流程图如下：



规定进程信息文件中描述进程的格式（使用空格分割）：

```
进程名称 到达时间 区间时间
```

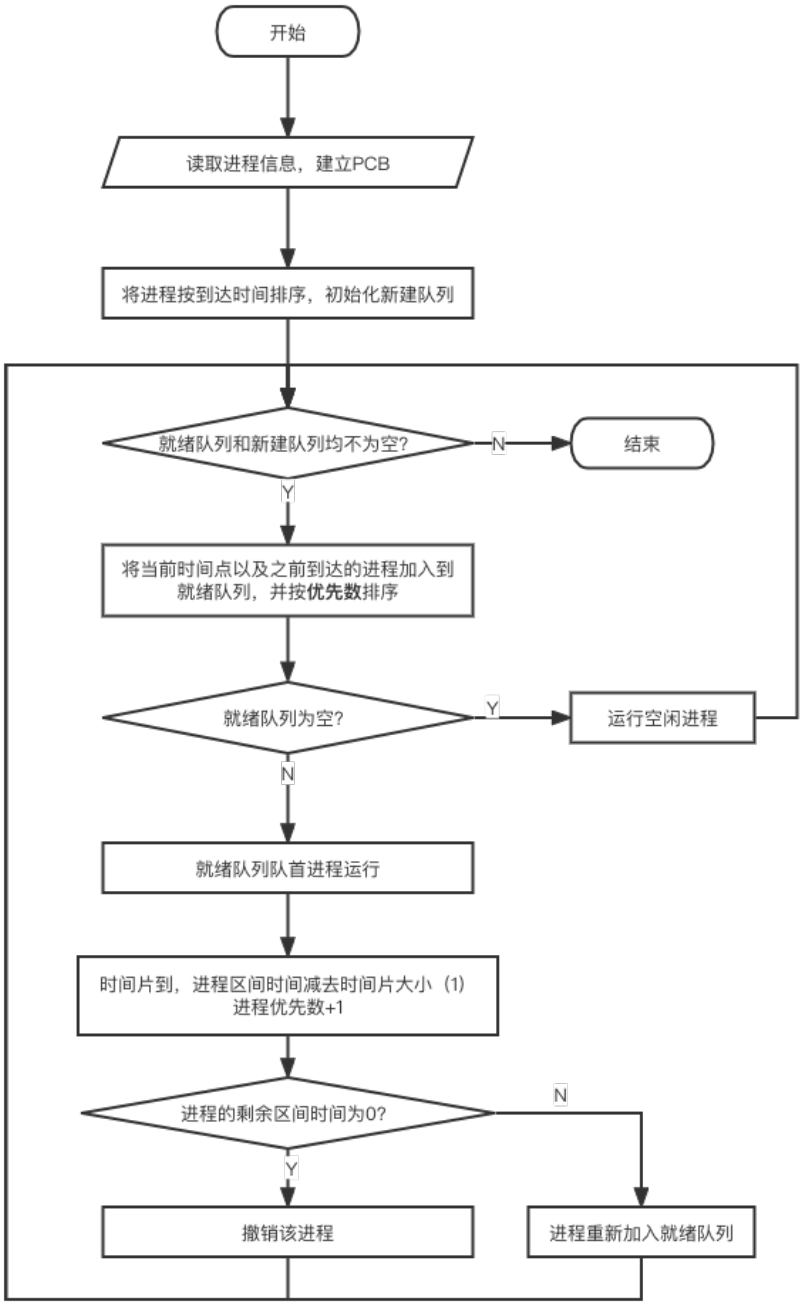
代码如下：

```
/* schedule/schedule_srtf.c */
#include <stdio.h>
#include <stdlib.h>
#include "schedule_proc.h"
// 按到达时间、区间时间比较进程 PCB 函数、进程读取函数与 SJF 相同，在此省略
/**
 * @brief 判断新进程是否抢占正在运行的进程，并将新进程加入就绪队列：
 * 若可以抢占，则正在运行的进程和新进程均加入就绪队列，均设置为就绪状态；
 * 否则，仅将新进程加入就绪队列，设置为就绪状态。
 * 不会对就绪队列进行排序，原因是 main 函数每次循环都会执行排序。
 * @param running_proc 正在运行的进程的 PCB
 * @param new_proc 新进程的 PCB
 * @return 0 - 不会抢占；!0 - 会抢占
 */
int interrupt(pcb *running_proc, pcb *new_proc) {
  if (new_proc->interval_time < running_proc->interval_time) {
    // 比当前进程的剩余时间更短（严格小于）的进程到达，会抢占
    push_ready(new_proc);
    printf("有新进程 %s 到达：\n", new_proc->name);
    display_proc(new_proc);
    push_ready(running_proc);
    printf("当前进程 %s 被抢占\n", running_proc->name);
    return !0;
  } else {  // 不会抢占
    push_ready(new_proc);
    printf("有新进程 %s 到达：\n", new_proc->name);
    display_proc(new_proc);
    return 0;
  }
}
/**
 * @brief 每运行一个 CPU 时间后检查新建队列中是否有新进程到达，
 * 若有则转向"中断"判断是否抢占
 * @param proc 进程 PCB
 */
void run(pcb *proc) {
  if (NULL != proc) {
    proc->states = PROC_STATE_RUNNING;  // 设置进程运行状态
    printf("当前正在运行的进程是 %s：\n", proc->name); // 当前运行的进程
    display_proc(proc);
    long interval = 0;
    while (proc->interval_time > 0) {
```

```
    proc->interval_time--;  // 运行一个 CPU 时间
    machine_time++;
    interval++;
    if (NULL!=created && created->arrive_time<=machine_time) {
      pcb *new_proc = created;  // 有新进程到达
      created = created->next_proc;
      new_proc->next_proc = NULL;
      if (0 != interrupt(proc, new_proc)) { break; }
    }
  }
  printf("进程 %s 运行时间: %ld\n", proc->name, interval);
  if (proc->interval_time == 0) {
    proc->states = PROC_STATE_FINISH;  // 结束
    proc->finish_time = machine_time;
    printf("进程 %s 运行结束: \n", proc->name);
    display_proc(proc);
  }
 }
}
int main(int argc, const char **argv) {
 if (argc != 2) {
  fprintf(stderr, "请提供一个命令行参数作为被调度进程信息! \n");
  return EXIT_FAILURE;
 }
 if (EXIT_SUCCESS != initialize(argv[1])) {
  fprintf(stderr, "文件读取失败! \n");
  return EXIT_FAILURE;
 }
 pcb *proc = NULL;
 while (NULL != ready || NULL != created) {//两队列均不为空才调度完成
  new_proc_ready();  // 将当前时间点以及之前到达的 PCB 加入到就绪队列
  // 按作业长短对就绪队列进行排序
  ready = sort_list(ready, compare_shortest_job);
  reset_ready_tail();  display_ready();
  if (NULL == ready) { // 若就绪队列为空，说明当前时间 CPU 空闲
    idle_proc(created->arrive_time - machine_time);
    continue;
  }
  proc = pop_ready();  // 就绪队列队首出队
  run(proc);  // 运行
  if (PROC_STATE_FINISH == proc->states) { // 运行结束，回收进程资源
    destroy_proc(proc);
  }
 }
 printf("\n*** 所有进程调度完成 ***\n");
```

```
  return EXIT_SUCCESS;
}
```

(5) 动态优先级（PR）：假定低优先数，高优先级，每次调度优先数最大的进程。被调度的进程每次只能运行一个 CPU 时间，运行后若未结束，将优先数+1（即优先级降低 1），重新加入就绪队列；否则撤销该进程。重新加入就绪队列的进程不改变到达时间。如果就绪队列出现优先数一样大的，则优先调度到达时间小的。若有比当前进程的优先数更大（严格大于）的进程到达时，则会抢占当前进程。这是一种抢占式的调度。使用 PR 算法的调度程序的流程图如下：



规定进程信息文件中描述进程的格式（使用空格分割）：

```
进程名称 优先数 到达时间 区间时间
```

实现中对于最后一个要调度的进程做了特殊处理，分配给其全部剩余区间时间的时间片，以减少调度次数。代码如下：

```
/* schedule/schedule_pr.c */
#include <stdio.h>
#include <stdlib.h>
#include "schedule_proc.h"
/** 按到达时间比较进程 PCB */
long compare_early_arrive(pcb *p1, pcb *p2) {
  return p1->arrive_time - p2->arrive_time;
}
/** 按优先数比较进程 PCB */
long compare_minimal_priority(pcb *p1, pcb *p2) {
  if (p1->priority == p2->priority) {
    return compare_early_arrive(p1, p2);
  }
  return p1->priority - p2->priority;
}
/** 进程读取函数 */
int initialize(const char *filename) {
  FILE *fp = fopen(filename, "r");
  if (NULL == fp) { return EXIT_FAILURE; }
  int count;
  pcb *p = NULL;
  fscanf(fp, "%d", &count);
  for (unsigned pid = 1; pid <= count; ++pid) {
    char name[16]; int priority; long arrive, interval;
    fscanf(fp, "%s %d %ld %ld",
           name, &priority, &arrive, &interval);
    if (NULL == created) {  // 在新建队列等候
      created = create_proc(pid, name, priority, arrive, interval);
      p = created;
    } else {
      p->next_proc=create_proc(pid,name,priority,arrive,interval);
      p = p->next_proc;
    }
  }
  created=sort_list(created, compare_early_arrive); //按到达时间排序
  return EXIT_SUCCESS;
}
/**
 * @brief PR，仅运行一个 CPU 时间
 * @param proc 进程 PCB
 */
void run(pcb *proc) {
  if (NULL != proc) {
    proc->states = PROC_STATE_RUNNING;  // 设置进程运行状态
    printf("当前正在运行的进程是 %s: \n", proc->name); // 当前运行的进程
```

```
    display_proc(proc);
    long interval = 1;
    if (NULL == ready && NULL == created) {  // 当前是最后一个进程
      interval = proc->interval_time;  // 运行剩余的 CPU 时间
      proc->interval_time = 0;
      proc->priority += (int) interval;  // 运行一个 CPU 时间，优先数+1
      machine_time += interval;
    } else {
      proc->interval_time--;  // 运行一个 CPU 时间
      proc->priority++;  // 每运行一个 CPU 时间，优先数+1
      machine_time++;
    }
    printf("进程 %s 运行时间：%ld，优先级变为：%d\n",
           proc->name, interval, proc->priority);
    if (0 == proc->interval_time) {
      proc->states = PROC_STATE_FINISH;  // 结束
      proc->finish_time = machine_time;
      printf("进程 %s 运行结束：\n", proc->name);
      display_proc(proc);
    } else {
      push_ready(proc);  // 加入就绪队列
    }
  }
}
int main(int argc, const char **argv) {
  if (argc != 2) {
    fprintf(stderr, "请提供一个命令行参数作为被调度进程信息！\n");
    return EXIT_FAILURE;
  }
  if (EXIT_SUCCESS != initialize(argv[1])) {
    fprintf(stderr, "文件读取失败！\n");
    return EXIT_FAILURE;
  }
  pcb *proc = NULL;
  while (NULL != ready || NULL != created){ //两队列均不为空才调度完成
    new_proc_ready();  // 将当前时间点以及之前到达的 PCB 加入到就绪队列
    // 按优先数对就绪队列进行排序
    ready = sort_list(ready, compare_minimal_priority);
    reset_ready_tail();  display_ready();
    if (NULL == ready) { // 若就绪队列为空，说明当前时间 CPU 空闲
      idle_proc(created->arrive_time - machine_time);
      continue;
    }
    proc = pop_ready();  // 就绪队列队首出队
    run(proc);  // 运行
```

```
    if (PROC_STATE_FINISH == proc->states) { // 运行结束，回收进程资源
      destroy_proc(proc);
    }
  }
  printf("\n*** 所有进程调度完成 ***\n");
  return EXIT_SUCCESS;
}
```

四. 实验结果和分析

1. （**实验 3.1：进程的创建**）

   代码位于 fork 目录下，使用 make 编译。

   验证 `fork()`：

```
parallels@ubuntu:~/codes/exp02$ ./process-fork
[PARENT] 当前处于父进程中
[PARENT] 父进程 PID = 37470
[PARENT] 子进程 PID = 37471
[PARENT] 父进程将等待子进程运行结束...
[CHILD] 当前处于子进程中
[CHILD] 子进程 PID = 37471
[CHILD] 父进程 PID = 37470
[CHILD] 子进程睡眠 1 秒...
[CHILD] 请输入子进程执行完毕后的返回值(0-255): 2
[CHILD] 子进程即将退出
[PARENT] 子进程的返回值: 2
[PARENT] 父进程即将退出
```

   在父进程获取到子进程的返回值之前，父进程和子进程执行的顺序是不确定的，如：

```
parallels@ubuntu:~/codes/exp02$ ./process-fork
[PARENT] 当前处于父进程中
[PARENT] 父进程 PID = 239776
[CHILD] 当前处于子进程中
[PARENT] 子进程 PID = 239777
[PARENT] 父进程将等待子进程运行结束...
[CHILD] 子进程 PID = 239777
[CHILD] 父进程 PID = 239776
[CHILD] 子进程睡眠 1 秒...
[CHILD] 请输入子进程执行完毕后的返回值(0-255): 2
[CHILD] 子进程即将退出
[PARENT] 子进程的返回值: 2
[PARENT] 父进程即将退出
```

```
parallels@ubuntu:~/codes/exp02$ ./process-fork
[PARENT] 当前处于父进程中
[CHILD] 当前处于子进程中
[PARENT] 父进程 PID = 240174
[CHILD] 子进程 PID = 240175
[CHILD] 父进程 PID = 240174
[CHILD] 子进程睡眠 1 秒...
[PARENT] 子进程 PID = 240175
[PARENT] 父进程将等待子进程运行结束...
[CHILD] 请输入子进程执行完毕后的返回值(0-255): 2
[CHILD] 子进程即将退出
[PARENT] 子进程的返回值: 2
[PARENT] 父进程即将退出
```

   使用 exec 函数族执行命令 ls：

```
parallels@ubuntu:~/codes/exp02$ gcc process-exec.c -o process-exec
parallels@ubuntu:~/codes/exp02$ ./process-exec
[PARENT] 当前处于父进程中
[PARENT] 父进程将等待子进程运行结束...
[CHILD] 当前处于子进程中
total 72K
drwxrwxr-x 3 parallels parallels 4.0K Mar 10 12:49 .
drwxrwxr-x 3 parallels parallels 4.0K Mar 10 10:29 ..
-rw-rw-r-- 1 parallels parallels    0 Mar 10 10:41 .clion.source.upload.marker
drwxrwxr-x 4 parallels parallels 4.0K Mar 10 12:44 cmake-build-debug
-rw-rw-r-- 1 parallels parallels  272 Mar 10 12:43 CMakeLists.txt
-rw-rw-r-- 1 parallels parallels   48 Mar 10 10:48 fcfs.c
-rw-rw-r-- 1 parallels parallels   76 Mar 10 10:31 main.c
-rw-rw-r-- 1 parallels parallels   48 Mar 10 10:48 pr.c
-rwxrwxr-x 1 parallels parallels  14K Mar 10 12:49 process-exec
-rw-rw-r-- 1 parallels parallels 1.2K Mar 10 12:39 process-exec.c
-rwxrwxr-x 1 parallels parallels  14K Mar 10 11:41 process-fork
-rw-rw-r-- 1 parallels parallels 1.4K Mar 10 12:37 process-fork.c
-rw-rw-r-- 1 parallels parallels   48 Mar 10 10:48 rr.c
[PARENT] 子进程的返回值: 0
[PARENT] 父进程即将退出
```

2. （**实验 3.2：进程调度算法的模拟**）

代码和测试数据位于 schedule 目录下，使用 make 编译。

(1) 先来先服务（FCFS）

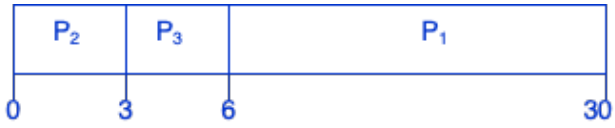
测试用例为 example-fcfs.txt，调度 3 个进程，其信息为：（名称 到达时间 区间时间）

```
P1 3 24
P2 0 3
P3 2 3
```

运行截图：

```
parallels@ubuntu:~/codes/exp02/schedule$ ./schedule_fcfs example-fcfs.txt

*** 就绪队列 (在处理机时间0) ***
PID: 2   PROCESS: P2     PRIOR: 1        STATES: READY          Arrive: 0      Interval: 3  Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P2:
PID: 2   PROCESS: P2     PRIOR: 1        STATES: RUNNING        Arrive: 0      Interval: 3  Finish: -1
进程 P2 运行时间：3
进程 P2 运行结束：
PID: 2   PROCESS: P2     PRIOR: 1        STATES: FINISH         Arrive: 0      Interval: 3  Finish: 3

*** 就绪队列 (在处理机时间3) ***
PID: 3   PROCESS: P3     PRIOR: 1        STATES: READY          Arrive: 2      Interval: 3  Finish: -1
PID: 1   PROCESS: P1     PRIOR: 1        STATES: READY          Arrive: 3      Interval: 24        Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P3:
PID: 3   PROCESS: P3     PRIOR: 1        STATES: RUNNING        Arrive: 2      Interval: 3  Finish: -1
进程 P3 运行时间：3
进程 P3 运行结束：
PID: 3   PROCESS: P3     PRIOR: 1        STATES: FINISH         Arrive: 2      Interval: 3  Finish: 6

*** 就绪队列 (在处理机时间6) ***
PID: 1   PROCESS: P1     PRIOR: 1        STATES: READY          Arrive: 3      Interval: 24        Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P1:
PID: 1   PROCESS: P1     PRIOR: 1        STATES: RUNNING        Arrive: 3      Interval: 24        Finish: -1
进程 P1 运行时间：24
进程 P1 运行结束：
PID: 1   PROCESS: P1     PRIOR: 1        STATES: FINISH         Arrive: 3      Interval: 24        Finish: 30

*** 所有进程调度完成 ***
```

使用甘特图表示调度流程为：



(2) 时间片轮转（RR）

测试用例为 example-rr.txt，调度 4 个进程，其信息为：（名称 到达时间 区间时间）

```
P1 0 23
P2 1 17
P3 2 46
P4 3 24
```

运行截图（不完整）：

```
parallels@ubuntu:~/codes/exp02/schedule$ ./schedule_rr example-rr.txt

*** 就绪队列 (在处理机时间0) ***
PID: 1   PROCESS: P1     PRIOR: 1        STATES: READY          Arrive: 0      Interval: 23    Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P1：
PID: 1   PROCESS: P1     PRIOR: 1        STATES: RUNNING        Arrive: 0      Interval: 23    Finish: -1
进程 P1 运行时间：20
进程 P1 被抢占

*** 就绪队列 (在处理机时间20) ***
PID: 2   PROCESS: P2     PRIOR: 1        STATES: READY          Arrive: 1      Interval: 17    Finish: -1
PID: 3   PROCESS: P3     PRIOR: 1        STATES: READY          Arrive: 2      Interval: 46    Finish: -1
PID: 4   PROCESS: P4     PRIOR: 1        STATES: READY          Arrive: 3      Interval: 24    Finish: -1
PID: 1   PROCESS: P1     PRIOR: 1        STATES: READY          Arrive: 20     Interval: 3     Finish: -1
*** 就绪队列进程数量 4 ***

当前正在运行的进程是 P2：
PID: 2   PROCESS: P2     PRIOR: 1        STATES: RUNNING        Arrive: 1      Interval: 17    Finish: -1
进程 P2 运行时间：17
进程 P2 运行结束：
PID: 2   PROCESS: P2     PRIOR: 1        STATES: FINISH         Arrive: 1      Interval: 17    Finish: 37
```

模拟程序输出的调度流程：

```
*** 就绪队列 (在处理机时间 0) ***
PID: 1  PROCESS: P1  PRIOR: 1  STATES: READY    Arrive: 0
Interval: 23   Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P1:
PID: 1  PROCESS: P1  PRIOR: 1  STATES: RUNNING     Arrive: 0
Interval: 23   Finish: -1
进程 P1 运行时间：20
进程 P1 被抢占

*** 就绪队列 (在处理机时间 20) ***
PID: 2  PROCESS: P2  PRIOR: 1  STATES: READY    Arrive: 1
Interval: 17   Finish: -1
PID: 3  PROCESS: P3  PRIOR: 1  STATES: READY    Arrive: 2
Interval: 46   Finish: -1
PID: 4  PROCESS: P4  PRIOR: 1  STATES: READY    Arrive: 3
Interval: 24   Finish: -1
PID: 1  PROCESS: P1  PRIOR: 1  STATES: READY    Arrive: 20
Interval: 3    Finish: -1
*** 就绪队列进程数量 4 ***

当前正在运行的进程是 P2:
PID: 2  PROCESS: P2  PRIOR: 1  STATES: RUNNING     Arrive: 1
Interval: 17   Finish: -1
```

```
进程 P2 运行时间：17
进程 P2 运行结束：
PID: 2  PROCESS: P2  PRIOR: 1  STATES: FINISH  Arrive: 1
Interval: 17  Finish: 37

*** 就绪队列 (在处理机时间 37) ***
PID: 3  PROCESS: P3  PRIOR: 1  STATES: READY  Arrive: 2
Interval: 46  Finish: -1
PID: 4  PROCESS: P4  PRIOR: 1  STATES: READY  Arrive: 3
Interval: 24  Finish: -1
PID: 1  PROCESS: P1  PRIOR: 1  STATES: READY  Arrive: 20
Interval: 3  Finish: -1
*** 就绪队列进程数量 3 ***

当前正在运行的进程是 P3：
PID: 3  PROCESS: P3  PRIOR: 1  STATES: RUNNING  Arrive: 2
Interval: 46  Finish: -1
进程 P3 运行时间：20
进程 P3 被抢占

*** 就绪队列 (在处理机时间 57) ***
PID: 4  PROCESS: P4  PRIOR: 1  STATES: READY  Arrive: 3
Interval: 24  Finish: -1
PID: 1  PROCESS: P1  PRIOR: 1  STATES: READY  Arrive: 20
Interval: 3  Finish: -1
PID: 3  PROCESS: P3  PRIOR: 1  STATES: READY  Arrive: 57
Interval: 26  Finish: -1
*** 就绪队列进程数量 3 ***

当前正在运行的进程是 P4：
PID: 4  PROCESS: P4  PRIOR: 1  STATES: RUNNING  Arrive: 3
Interval: 24  Finish: -1
进程 P4 运行时间：20
进程 P4 被抢占

*** 就绪队列 (在处理机时间 77) ***
PID: 1  PROCESS: P1  PRIOR: 1  STATES: READY  Arrive: 20
Interval: 3  Finish: -1
PID: 3  PROCESS: P3  PRIOR: 1  STATES: READY  Arrive: 57
Interval: 26  Finish: -1
PID: 4  PROCESS: P4  PRIOR: 1  STATES: READY  Arrive: 77
Interval: 4  Finish: -1
*** 就绪队列进程数量 3 ***

当前正在运行的进程是 P1：
```

```
PID: 1  PROCESS: P1  PRIOR: 1  STATES: RUNNING     Arrive: 20
Interval: 3    Finish: -1
进程 P1 运行时间：3
进程 P1 运行结束：
PID: 1  PROCESS: P1  PRIOR: 1  STATES: FINISH   Arrive: 20
Interval: 3    Finish: 80

*** 就绪队列 (在处理机时间 80) ***
PID: 3  PROCESS: P3  PRIOR: 1  STATES: READY    Arrive: 57
Interval: 26   Finish: -1
PID: 4  PROCESS: P4  PRIOR: 1  STATES: READY    Arrive: 77
Interval: 4    Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P3：
PID: 3  PROCESS: P3  PRIOR: 1  STATES: RUNNING     Arrive: 57
Interval: 26   Finish: -1
进程 P3 运行时间：20
进程 P3 被抢占

*** 就绪队列 (在处理机时间 100) ***
PID: 4  PROCESS: P4  PRIOR: 1  STATES: READY    Arrive: 77
Interval: 4    Finish: -1
PID: 3  PROCESS: P3  PRIOR: 1  STATES: READY    Arrive: 100
Interval: 6    Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P4：
PID: 4  PROCESS: P4  PRIOR: 1  STATES: RUNNING     Arrive: 77
Interval: 4    Finish: -1
进程 P4 运行时间：4
进程 P4 运行结束：
PID: 4  PROCESS: P4  PRIOR: 1  STATES: FINISH   Arrive: 77
Interval: 4    Finish: 104

*** 就绪队列 (在处理机时间 104) ***
PID: 3  PROCESS: P3  PRIOR: 1  STATES: READY    Arrive: 100
Interval: 6    Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P3：
PID: 3  PROCESS: P3  PRIOR: 1  STATES: RUNNING     Arrive: 100
Interval: 6    Finish: -1
进程 P3 运行时间：6
进程 P3 运行结束：
```

```
PID: 3   PROCESS: P3   PRIOR: 1   STATES: FINISH   Arrive: 100
Interval: 6    Finish: 110

*** 所有进程调度完成 ***
```

使用甘特图表示调度流程为：

(3) 短作业优先（SJF）

测试用例为 example-sjf_srtf.txt，调度 4 个进程，其信息为：（名称 到达时间 区间时间）

```
P1 0 7
P2 2 4
P3 4 1
P4 5 4
```

运行截图（不完整）：

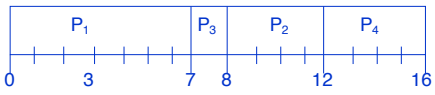
```
parallels@ubuntu:~/codes/exp02/schedule$ ./schedule_sjf example-sjf_srtf.txt

*** 就绪队列 (在处理机时间0) ***
PID: 1   PROCESS: P1     PRIOR: 1        STATES: READY          Arrive: 0      Interval: 7    Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P1：
PID: 1   PROCESS: P1     PRIOR: 1        STATES: RUNNING        Arrive: 0      Interval: 7    Finish: -1
进程 P1 运行时间：7
进程 P1 运行结束：
PID: 1   PROCESS: P1     PRIOR: 1        STATES: FINISH         Arrive: 0      Interval: 7    Finish: 7

*** 就绪队列 (在处理机时间7) ***
PID: 3   PROCESS: P3     PRIOR: 1        STATES: READY          Arrive: 4      Interval: 1    Finish: -1
PID: 2   PROCESS: P2     PRIOR: 1        STATES: READY          Arrive: 2      Interval: 4    Finish: -1
PID: 4   PROCESS: P4     PRIOR: 1        STATES: READY          Arrive: 5      Interval: 4    Finish: -1
*** 就绪队列进程数量 3 ***

当前正在运行的进程是 P3：
PID: 3   PROCESS: P3     PRIOR: 1        STATES: RUNNING        Arrive: 4      Interval: 1    Finish: -1
进程 P3 运行时间：1
进程 P3 运行结束：
PID: 3   PROCESS: P3     PRIOR: 1        STATES: FINISH         Arrive: 4      Interval: 1    Finish: 8
```

模拟程序输出的调度流程：

```
*** 就绪队列 (在处理机时间 0) ***
PID: 1   PROCESS: P1   PRIOR: 1   STATES: READY    Arrive: 0
Interval: 7    Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P1：
PID: 1   PROCESS: P1   PRIOR: 1   STATES: RUNNING     Arrive: 0
Interval: 7    Finish: -1
进程 P1 运行时间：7
进程 P1 运行结束：
PID: 1   PROCESS: P1   PRIOR: 1   STATES: FINISH   Arrive: 0
Interval: 7    Finish: 7

*** 就绪队列 (在处理机时间 7) ***
PID: 3   PROCESS: P3   PRIOR: 1   STATES: READY    Arrive: 4
Interval: 1    Finish: -1
PID: 2   PROCESS: P2   PRIOR: 1   STATES: READY    Arrive: 2
```

```
Interval: 4    Finish: -1
PID: 4  PROCESS: P4  PRIOR: 1  STATES: READY    Arrive: 5
Interval: 4    Finish: -1
*** 就绪队列进程数量 3 ***

当前正在运行的进程是 P3:
PID: 3  PROCESS: P3  PRIOR: 1  STATES: RUNNING     Arrive: 4
Interval: 1    Finish: -1
进程 P3 运行时间: 1
进程 P3 运行结束:
PID: 3  PROCESS: P3  PRIOR: 1  STATES: FINISH   Arrive: 4
Interval: 1    Finish: 8

*** 就绪队列 (在处理机时间 8) ***
PID: 2  PROCESS: P2  PRIOR: 1  STATES: READY    Arrive: 2
Interval: 4    Finish: -1
PID: 4  PROCESS: P4  PRIOR: 1  STATES: READY    Arrive: 5
Interval: 4    Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P2:
PID: 2  PROCESS: P2  PRIOR: 1  STATES: RUNNING     Arrive: 2
Interval: 4    Finish: -1
进程 P2 运行时间: 4
进程 P2 运行结束:
PID: 2  PROCESS: P2  PRIOR: 1  STATES: FINISH   Arrive: 2
Interval: 4    Finish: 12

*** 就绪队列 (在处理机时间 12) ***
PID: 4  PROCESS: P4  PRIOR: 1  STATES: READY    Arrive: 5
Interval: 4    Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P4:
PID: 4  PROCESS: P4  PRIOR: 1  STATES: RUNNING     Arrive: 5
Interval: 4    Finish: -1
进程 P4 运行时间: 4
进程 P4 运行结束:
PID: 4  PROCESS: P4  PRIOR: 1  STATES: FINISH   Arrive: 5
Interval: 4    Finish: 16

*** 所有进程调度完成 ***
```

使用甘特图表示调度流程为：



(4) 最短剩余时间优先（SRTF）

测试用例同 SJF，为 example-sjf_srtf.txt，调度 4 个进程。

运行截图（不完整）：

```
parallels@ubuntu:~/codes/exp02/schedule$ ./schedule_srtf example-sjf_srtf.txt

*** 就绪队列 (在处理机时间0) ***
PID: 1   PROCESS: P1     PRIOR: 1        STATES: READY          Arrive: 0      Interval: 7     Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P1：
PID: 1   PROCESS: P1     PRIOR: 1        STATES: RUNNING        Arrive: 0      Interval: 7     Finish: -1
有新进程 P2 到达：
PID: 2   PROCESS: P2     PRIOR: 1        STATES: READY          Arrive: 2      Interval: 4     Finish: -1
当前进程 P1 被抢占
进程 P1 运行时间：2

*** 就绪队列 (在处理机时间2) ***
PID: 2   PROCESS: P2     PRIOR: 1        STATES: READY          Arrive: 2      Interval: 4     Finish: -1
PID: 1   PROCESS: P1     PRIOR: 1        STATES: READY          Arrive: 0      Interval: 5     Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P2：
PID: 2   PROCESS: P2     PRIOR: 1        STATES: RUNNING        Arrive: 2      Interval: 4     Finish: -1
有新进程 P3 到达：
PID: 3   PROCESS: P3     PRIOR: 1        STATES: READY          Arrive: 4      Interval: 1     Finish: -1
当前进程 P2 被抢占
进程 P2 运行时间：2
```

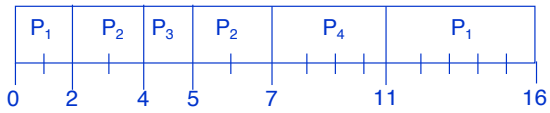
模拟程序输出的调度流程：

```
*** 就绪队列 (在处理机时间 0) ***
PID: 1  PROCESS: P1  PRIOR: 1  STATES: READY    Arrive: 0
Interval: 7    Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P1：
PID: 1  PROCESS: P1  PRIOR: 1  STATES: RUNNING     Arrive: 0
Interval: 7    Finish: -1
有新进程 P2 到达：
PID: 2  PROCESS: P2  PRIOR: 1  STATES: READY    Arrive: 2
Interval: 4    Finish: -1
当前进程 P1 被抢占
进程 P1 运行时间：2

*** 就绪队列 (在处理机时间 2) ***
PID: 2  PROCESS: P2  PRIOR: 1  STATES: READY    Arrive: 2
Interval: 4    Finish: -1
PID: 1  PROCESS: P1  PRIOR: 1  STATES: READY    Arrive: 0
Interval: 5    Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P2：
PID: 2  PROCESS: P2  PRIOR: 1  STATES: RUNNING     Arrive: 2
Interval: 4    Finish: -1
有新进程 P3 到达：
PID: 3  PROCESS: P3  PRIOR: 1  STATES: READY    Arrive: 4
Interval: 1    Finish: -1
当前进程 P2 被抢占
```

```
进程 P2 运行时间：2

*** 就绪队列 (在处理机时间 4) ***
PID: 3  PROCESS: P3  PRIOR: 1  STATES: READY    Arrive: 4
Interval: 1    Finish: -1
PID: 2  PROCESS: P2  PRIOR: 1  STATES: READY    Arrive: 2
Interval: 2    Finish: -1
PID: 1  PROCESS: P1  PRIOR: 1  STATES: READY    Arrive: 0
Interval: 5    Finish: -1
*** 就绪队列进程数量 3 ***

当前正在运行的进程是 P3：
PID: 3  PROCESS: P3  PRIOR: 1  STATES: RUNNING     Arrive: 4
Interval: 1    Finish: -1
有新进程 P4 到达：
PID: 4  PROCESS: P4  PRIOR: 1  STATES: READY    Arrive: 5
Interval: 4    Finish: -1
进程 P3 运行时间：1
进程 P3 运行结束：
PID: 3  PROCESS: P3  PRIOR: 1  STATES: FINISH   Arrive: 4
Interval: 0    Finish: 5

*** 就绪队列 (在处理机时间 5) ***
PID: 2  PROCESS: P2  PRIOR: 1  STATES: READY    Arrive: 2
Interval: 2    Finish: -1
PID: 4  PROCESS: P4  PRIOR: 1  STATES: READY    Arrive: 5
Interval: 4    Finish: -1
PID: 1  PROCESS: P1  PRIOR: 1  STATES: READY    Arrive: 0
Interval: 5    Finish: -1
*** 就绪队列进程数量 3 ***

当前正在运行的进程是 P2：
PID: 2  PROCESS: P2  PRIOR: 1  STATES: RUNNING     Arrive: 2
Interval: 2    Finish: -1
进程 P2 运行时间：2
进程 P2 运行结束：
PID: 2  PROCESS: P2  PRIOR: 1  STATES: FINISH   Arrive: 2
Interval: 0    Finish: 7

*** 就绪队列 (在处理机时间 7) ***
PID: 4  PROCESS: P4  PRIOR: 1  STATES: READY    Arrive: 5
Interval: 4    Finish: -1
PID: 1  PROCESS: P1  PRIOR: 1  STATES: READY    Arrive: 0
Interval: 5    Finish: -1
*** 就绪队列进程数量 2 ***
```

```
当前正在运行的进程是 P4：
PID: 4   PROCESS: P4   PRIOR: 1   STATES: RUNNING     Arrive: 5
Interval: 4    Finish: -1
进程 P4 运行时间：4
进程 P4 运行结束：
PID: 4   PROCESS: P4   PRIOR: 1   STATES: FINISH   Arrive: 5
Interval: 0    Finish: 11

*** 就绪队列 (在处理机时间 11) ***
PID: 1   PROCESS: P1   PRIOR: 1   STATES: READY    Arrive: 0
Interval: 5    Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P1：
PID: 1   PROCESS: P1   PRIOR: 1   STATES: RUNNING     Arrive: 0
Interval: 5    Finish: -1
进程 P1 运行时间：5
进程 P1 运行结束：
PID: 1   PROCESS: P1   PRIOR: 1   STATES: FINISH   Arrive: 0
Interval: 0    Finish: 16

*** 所有进程调度完成 ***
```

使用甘特图表示调度流程为：



(5) 动态优先级（PR）

测试用例为 example-pr.txt，调度 5 个进程，其信息为:（名称 优先数 到达时间 区间时间）

```
P1 3 0 10
P2 1 1 1
P3 3 2 2
P4 4 3 1
P5 2 4 5
```

运行截图（不完整）：

```
parallels@ubuntu:~/codes/exp02/schedule$ ./schedule_pr example-pr.txt

*** 就绪队列 (在处理机时间0) ***
PID: 1   PROCESS: P1     PRIOR: 3        STATES: READY          Arrive: 0      Interval: 10    Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P1：
PID: 1   PROCESS: P1     PRIOR: 3        STATES: RUNNING        Arrive: 0      Interval: 10    Finish: -1
进程 P1 运行时间：1, 优先级变为：4

*** 就绪队列 (在处理机时间1) ***
PID: 2   PROCESS: P2     PRIOR: 1        STATES: READY          Arrive: 1      Interval: 1     Finish: -1
PID: 1   PROCESS: P1     PRIOR: 4        STATES: READY          Arrive: 0      Interval: 9     Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P2：
PID: 2   PROCESS: P2     PRIOR: 1        STATES: RUNNING        Arrive: 1      Interval: 1     Finish: -1
进程 P2 运行时间：1, 优先级变为：2
进程 P2 运行结束：
PID: 2   PROCESS: P2     PRIOR: 2        STATES: FINISH         Arrive: 1      Interval: 0     Finish: 2
```

模拟程序输出的调度流程：

```
*** 就绪队列 (在处理机时间 0) ***
PID: 1  PROCESS: P1  PRIOR: 3  STATES: READY    Arrive: 0
Interval: 10  Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P1：
PID: 1  PROCESS: P1  PRIOR: 3  STATES: RUNNING    Arrive: 0
Interval: 10  Finish: -1
进程 P1 运行时间：1，优先级变为：4

*** 就绪队列 (在处理机时间 1) ***
PID: 2  PROCESS: P2  PRIOR: 1  STATES: READY    Arrive: 1
Interval: 1   Finish: -1
PID: 1  PROCESS: P1  PRIOR: 4  STATES: READY    Arrive: 0
Interval: 9   Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P2：
PID: 2  PROCESS: P2  PRIOR: 1  STATES: RUNNING    Arrive: 1
Interval: 1   Finish: -1
进程 P2 运行时间：1，优先级变为：2
进程 P2 运行结束：
PID: 2  PROCESS: P2  PRIOR: 2  STATES: FINISH   Arrive: 1
Interval: 0   Finish: 2

*** 就绪队列 (在处理机时间 2) ***
PID: 3  PROCESS: P3  PRIOR: 3  STATES: READY    Arrive: 2
Interval: 2   Finish: -1
PID: 1  PROCESS: P1  PRIOR: 4  STATES: READY    Arrive: 0
Interval: 9   Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P3：
PID: 3  PROCESS: P3  PRIOR: 3  STATES: RUNNING    Arrive: 2
Interval: 2   Finish: -1
进程 P3 运行时间：1，优先级变为：4

*** 就绪队列 (在处理机时间 3) ***
PID: 1  PROCESS: P1  PRIOR: 4  STATES: READY    Arrive: 0
Interval: 9   Finish: -1
PID: 3  PROCESS: P3  PRIOR: 4  STATES: READY    Arrive: 2
Interval: 1   Finish: -1
PID: 4  PROCESS: P4  PRIOR: 4  STATES: READY    Arrive: 3
Interval: 1   Finish: -1
```

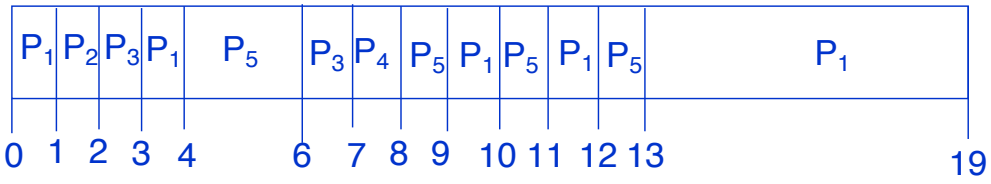
```
*** 就绪队列进程数量 3 ***

当前正在运行的进程是 P1：
PID: 1  PROCESS: P1  PRIOR: 4  STATES: RUNNING     Arrive: 0
Interval: 9    Finish: -1
进程 P1 运行时间：1，优先级变为：5

*** 就绪队列 (在处理机时间 4) ***
PID: 5  PROCESS: P5  PRIOR: 2  STATES: READY    Arrive: 4
Interval: 5    Finish: -1
PID: 3  PROCESS: P3  PRIOR: 4  STATES: READY    Arrive: 2
Interval: 1    Finish: -1
PID: 4  PROCESS: P4  PRIOR: 4  STATES: READY    Arrive: 3
Interval: 1    Finish: -1
PID: 1  PROCESS: P1  PRIOR: 5  STATES: READY    Arrive: 0
Interval: 8    Finish: -1
*** 就绪队列进程数量 4 ***

当前正在运行的进程是 P5：
PID: 5  PROCESS: P5  PRIOR: 2  STATES: RUNNING     Arrive: 4
Interval: 5    Finish: -1
进程 P5 运行时间：1，优先级变为：3

*** 就绪队列 (在处理机时间 5) ***
PID: 5  PROCESS: P5  PRIOR: 3  STATES: READY    Arrive: 4
Interval: 4    Finish: -1
PID: 3  PROCESS: P3  PRIOR: 4  STATES: READY    Arrive: 2
Interval: 1    Finish: -1
PID: 4  PROCESS: P4  PRIOR: 4  STATES: READY    Arrive: 3
Interval: 1    Finish: -1
PID: 1  PROCESS: P1  PRIOR: 5  STATES: READY    Arrive: 0
Interval: 8    Finish: -1
*** 就绪队列进程数量 4 ***

当前正在运行的进程是 P5：
PID: 5  PROCESS: P5  PRIOR: 3  STATES: RUNNING     Arrive: 4
Interval: 4    Finish: -1
进程 P5 运行时间：1，优先级变为：4

*** 就绪队列 (在处理机时间 6) ***
PID: 3  PROCESS: P3  PRIOR: 4  STATES: READY    Arrive: 2
Interval: 1    Finish: -1
PID: 4  PROCESS: P4  PRIOR: 4  STATES: READY    Arrive: 3
Interval: 1    Finish: -1
PID: 5  PROCESS: P5  PRIOR: 4  STATES: READY    Arrive: 4
```

```
Interval: 3    Finish: -1
PID: 1  PROCESS: P1  PRIOR: 5  STATES: READY    Arrive: 0
Interval: 8    Finish: -1
*** 就绪队列进程数量 4 ***

当前正在运行的进程是 P3：
PID: 3  PROCESS: P3  PRIOR: 4  STATES: RUNNING     Arrive: 2
Interval: 1    Finish: -1
进程 P3 运行时间：1，优先级变为：5
进程 P3 运行结束：
PID: 3  PROCESS: P3  PRIOR: 5  STATES: FINISH   Arrive: 2
Interval: 0    Finish: 7

*** 就绪队列 (在处理机时间 7) ***
PID: 4  PROCESS: P4  PRIOR: 4  STATES: READY    Arrive: 3
Interval: 1    Finish: -1
PID: 5  PROCESS: P5  PRIOR: 4  STATES: READY    Arrive: 4
Interval: 3    Finish: -1
PID: 1  PROCESS: P1  PRIOR: 5  STATES: READY    Arrive: 0
Interval: 8    Finish: -1
*** 就绪队列进程数量 3 ***

当前正在运行的进程是 P4：
PID: 4  PROCESS: P4  PRIOR: 4  STATES: RUNNING     Arrive: 3
Interval: 1    Finish: -1
进程 P4 运行时间：1，优先级变为：5
进程 P4 运行结束：
PID: 4  PROCESS: P4  PRIOR: 5  STATES: FINISH   Arrive: 3
Interval: 0    Finish: 8

*** 就绪队列 (在处理机时间 8) ***
PID: 5  PROCESS: P5  PRIOR: 4  STATES: READY    Arrive: 4
Interval: 3    Finish: -1
PID: 1  PROCESS: P1  PRIOR: 5  STATES: READY    Arrive: 0
Interval: 8    Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P5：
PID: 5  PROCESS: P5  PRIOR: 4  STATES: RUNNING     Arrive: 4
Interval: 3    Finish: -1
进程 P5 运行时间：1，优先级变为：5

*** 就绪队列 (在处理机时间 9) ***
PID: 1  PROCESS: P1  PRIOR: 5  STATES: READY    Arrive: 0
Interval: 8    Finish: -1
```

```
PID: 5  PROCESS: P5  PRIOR: 5  STATES: READY    Arrive: 4
Interval: 2    Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P1：
PID: 1  PROCESS: P1  PRIOR: 5  STATES: RUNNING    Arrive: 0
Interval: 8    Finish: -1
进程 P1 运行时间：1，优先级变为：6

*** 就绪队列 (在处理机时间 10) ***
PID: 5  PROCESS: P5  PRIOR: 5  STATES: READY    Arrive: 4
Interval: 2    Finish: -1
PID: 1  PROCESS: P1  PRIOR: 6  STATES: READY    Arrive: 0
Interval: 7    Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P5：
PID: 5  PROCESS: P5  PRIOR: 5  STATES: RUNNING    Arrive: 4
Interval: 2    Finish: -1
进程 P5 运行时间：1，优先级变为：6

*** 就绪队列 (在处理机时间 11) ***
PID: 1  PROCESS: P1  PRIOR: 6  STATES: READY    Arrive: 0
Interval: 7    Finish: -1
PID: 5  PROCESS: P5  PRIOR: 6  STATES: READY    Arrive: 4
Interval: 1    Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P1：
PID: 1  PROCESS: P1  PRIOR: 6  STATES: RUNNING    Arrive: 0
Interval: 7    Finish: -1
进程 P1 运行时间：1，优先级变为：7

*** 就绪队列 (在处理机时间 12) ***
PID: 5  PROCESS: P5  PRIOR: 6  STATES: READY    Arrive: 4
Interval: 1    Finish: -1
PID: 1  PROCESS: P1  PRIOR: 7  STATES: READY    Arrive: 0
Interval: 6    Finish: -1
*** 就绪队列进程数量 2 ***

当前正在运行的进程是 P5：
PID: 5  PROCESS: P5  PRIOR: 6  STATES: RUNNING    Arrive: 4
Interval: 1    Finish: -1
进程 P5 运行时间：1，优先级变为：7
进程 P5 运行结束：
```

```
PID: 5   PROCESS: P5   PRIOR: 7   STATES: FINISH   Arrive: 4
Interval: 0    Finish: 13

*** 就绪队列 (在处理机时间 13) ***
PID: 1   PROCESS: P1   PRIOR: 7   STATES: READY    Arrive: 0
Interval: 6    Finish: -1
*** 就绪队列进程数量 1 ***

当前正在运行的进程是 P1:
PID: 1   PROCESS: P1   PRIOR: 7   STATES: RUNNING     Arrive: 0
Interval: 6    Finish: -1
进程 P1 运行时间：6，优先级变为：13
进程 P1 运行结束:
PID: 1   PROCESS: P1   PRIOR: 13 STATES: FINISH   Arrive: 0
Interval: 0    Finish: 19

*** 所有进程调度完成 ***
```

使用甘特图表示调度流程为：



五. 讨论、心得

1. 本次实验的主要内容是进程创建和和调度模拟，具体为在 Linux 中使用 fork 函数以及 exec 函数族创建进程，以及编写 C 程序进行调度算法的模拟。通过本次实验，我进一步加深了对进程概念的理解，也对进程调度算法有了更深的认识。
2. 调度算法部分没有直接使用指导书中的示例代码，考虑了进程到达时间在调度中的影响，分别实现了先来先服务（FCFS）、时间片轮转（RR）、短作业优先（SJF）、最短剩余时间优先（SRTF）、动态优先级（PR）等调度算法，并使用上学期课件中的例子加以验证，并在最后使用甘特图来表示调度过程。在编码的使用过程中也遇到了一些小问题，借助帮助文档和网络资源可以很好的解决。
3. 总结调用 fork() 函数后的三种返回情况。
   (1) 返回值等于 0，表示当前进程是子进程，使用 getpid 函数可以获得子进程 PID，使用 getppid 函数可以获得父进程 PID。
   (2) 返回值大于 0，表示当前进程是父进程，值为子进程的 PID。
   (3) 返回值等于 0，表示进程创建失败。

   创建新进程成功后，系统中出现两个基本相同的进程，这两个进程执行没有固定的先后顺序，哪个进程先执行要看系统的进程调度策略。
4. 总结 fork() 和 wait() 配合使用的情况，并尝试在父进程中取消 wait() 函数，观察进程的运行情况。

   fork()调用后就会产生并运行子进程，当子进程退出时，内核会向父进程发送 SIGCHLD 信号，同时将子进程设置为僵死状态，而只保留最少的一些内核数据结构，以便父进程查询子进程

的状态。 而进程一旦调用了 wait()函数，就会立即阻塞自身，直到它的一个子进程结束为止，同时会收集这个子进程的信息，并在把它彻底销毁后返回。

若取消 wait()函数，则父进程会在运行结束后直接退出，不会有进程负责子进程的销毁，将一直保持在僵死状态。同时子进程的后续输出将不会出现。运行结果：

```
/root/tmp/tmp.nBDxEZe3wv/cmake-build-debug/process-fork
[PARENT] 当前处于父进程中
[PARENT] 父进程 PID = 87781
[PARENT] 子进程 PID = 87782
[PARENT] 父进程将等待子进程运行结束...
[PARENT] 子进程的返回值: 0
[PARENT] 父进程即将退出
[CHILD] 当前处于子进程中
[CHILD] 子进程 PID = 87782
[CHILD] 父进程 PID = 87781
[CHILD] 子进程睡眠 1 秒...
```

5. 总结 exec 函数族的具体使用方法如下。

exec 函数族中末尾的字母表示不同参数，其中：

- “l”和“v”表示参数是以可变参数列表还是以数组的方式提供，但是都要以 NULL 结尾，其中的第一个参数为需要执行二进制程序的名字；
- “p”表示这个函数的第一个参数是以绝对路径来提供程序的路径，否则作为文件名；
- “e”表示为程序提供新的环境变量，不需要以 NULL 结尾。