大连理工大学

本科实验报告

课程名称：操作系统实验

学院（系）：电子信息与电气工程学部

专　　业：计算机科学与技术啊

班　　级：电计 1402

学　　号：201774009

学生姓名：胡文博

2017 年 6 月 22 日

1. 实验一：进程管理

1.1 题目一

每个进程都执行自己独立的程序，打印自己的 pid，每个父进程打印其子进程的 pid;

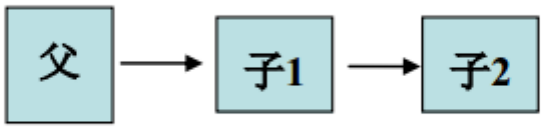

图 1.1

1.1.1 题目分析

题目的要求可分为三个：（1）父进程通过 fork 创建子进程，子进程再通过 fork 创建孙子进程（2）每个进程打印自己的 PID（3）父进程打印子进程 PID。所以进程可以通过 fork()函数创建子进程，fork()函数在父进程中的返回值是创建成功的子进程 PID 或者失败的负数返回值，fork()函数在子进程中的返回值是 0，故可通过返回值判定该进程是子进程还是父进程。打印自己的 PID 可通过 getpid()函数获得自身 PID 并打印。

1.1.2 代码实现

```
#include <iostream>
#include <unistd.h>
#include <cstdlib>
using namespace std;

int main(int argc, char const *argv[])
{
    pid_t pid;
    pid = fork();//创建子进程，并将返回值存入变量 pid
    if (pid < 0)//创建失败
    {
        cout << "creat son process error!" << endl;
        exit(-1);
    }
    else if ( pid == 0)//pid 为 0 代表该进程为子进程
    {
        pid_t pid2 = fork();//子进程创建孙子进程
```

```
        if (pid2 == 0)//pid2 为 0 代表该进程为孙子进程
        {
            cout << "I am the grandson process, my ID is: " << getpid() << endl;
        }
        else if (pid2 < 0)//创建失败
        {
            cout << "creat grandson process error!" << endl;
            exit(-1);
        }
        else//pid2 不为 0 代表该进程为孙子进程的父进程，即子进程
        {
            cout << "I am the son process, my ID is: " << getpid() << ", and my
son process (grandson process) ID is:" << pid2 << endl;
        }
    }
    else//pid 不为 0 代表该进程为父进程
    {
        cout << "I am the father process, my ID is: " << getpid() << ", and my
son process ID is:" << pid << endl;;
    }
    return 0;
}
```

1. 1. 3 实验结果

实验结果如图 1.2 所示，可见代码实现了实验要求。

1. 2 题目二

每个进程都执行自己独立的程序，打印自己的 pid，每个父进程打印其子进程的 pid;

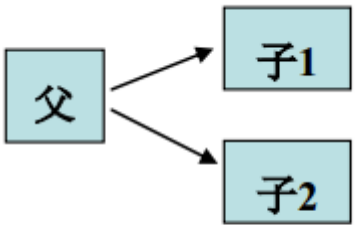

图 1.3

1. 2. 1 题目分析

此题与题目一的不同点在于父进程 fork()创建了两个子进程，而不是子进程创建孙子进程。

1.2.2 代码实现

```
/**
 *
 * @author:     胡文博
 * @email:      huwenbo@mail.dlut.edu.cn
 * @dateTime:       2017-05-15 18:37:23
 * @description:    father      ->   son1
                          ->   son2
 */

#include <iostream>
#include <unistd.h>
#include <cstdlib>

using namespace std;
int main(int argc, char const *argv[])
{
    pid_t pid;
    pid = fork();//创建子进程，并将返回值存入变量 pid
    if (pid < 0)//创建失败
    {
        cout << "create son1 process error!" << endl;
        exit(-1);
    }
    else if ( pid == 0)//pid 为 0 代表该进程为子进程
    {
        cout << "I am the son1 process, my ID is: " << getpid() << endl;
    }
    else//pid 不为 0 代表该进程为父进程
    {
        pid_t pid2 = fork();//父进程创建第二个子进程
        if (pid2 < 0)
        {
            cout << "creat son2 process error!" << endl;
            exit(-1);
        }
        else if (pid2 == 0)
        {
            cout << "I am the son2 process, my ID is: " << getpid() << endl;
        }
        else
        {
                cout << "I am the father process, my ID is: " << getpid() << ",
my first son ID is:"<<pid<<", and my second son process ID is:"<<pid2<<endl;
        }
    }
    return 0;
}
```

1.2.3 实验结果

实验结果如图 1.4 所示，可见代码实现了实验要求。

1.3 题目三

编写一个命令处理程序，能处理 max(m,n), min(m,n) 和 average(m,n,l)这几个命令。（使用 exec 函数族）

1.3.1 题目分析

首先需要编写程序编译出 max(m,n),min(m,n),average(m,n,l)这三个可执行文件，由于题目中未说明 m,n,l 的数据类型是整形还是浮点型，故做如下设定，若用户输入的数字含小数点则认为是浮点数，结果做浮点运算，否则按照整数运算。鉴于此前提，这三个可执行文件的编写采用模板函数来完成，代码见下节。

其次是命令处理程序，即 shell 程序的编写，我采用了正则表达式来进行词法解析，解析出命令和参数，然后根据命令使用 execlp()函数创建子进程执行相应命令。

1.3.2 代码实现

- max(m,n)的实现：

```
/**
 *
 * @author:     胡文博
 * @email:      huwenbo@mail.dlut.edu.cn
 * @dateTime:       2017-05-15 19:32:46
 * @description:
 */
#include <cstdlib>
#include <string>
#include <iostream>
#include <sstream>
using namespace std;

template <typename T>
inline T const& myMax (T const& a, T const& b)//模板函数实现任意数据类型的数据求最大值
{
    return  a < b ? b : a;
}
bool isInt(string s)//根据输入数据是否小数点来判断是整形数还是浮点数
{
    stringstream sin(s);
    char c;
    while(sin>>c)
    {
        if(c == '.')
            return false;
    }
    return true;
}
```

```
int main(int argc, char *argv[])
{
    if (argc != 3)//参数个数不对
    {
        cout << "the number of parameters is error!" << endl;
        exit(-1);
    }
    string s1 = argv[1], s2 = argv[2];
    if(isInt(s1) && isInt(s2))//整形输入
    {
        cout<<"among "<<s1<<" and "<<s2<<", "<<"the bigger is:
"<<myMax(atoi(s1.data()),atoi(s2.data()))<<endl;
    }
    else//浮点型输入
    {
        cout<<"among "<<s1<<" and "<<s2<<", "<<"the bigger is:
"<<myMax(atof(s1.data()),atof(s2.data()))<<endl;
    }
    return 0;
}
```

- min(m,n)的实现：

```
/**
 *
 * @author:     胡文博
 * @email:      huwenbo@mail.dlut.edu.cn
 * @dateTime:       2017-05-15 19:32:52
 * @description:
 */
#include <cstdlib>
#include <string>
#include <iostream>
#include <sstream>
using namespace std;

template <typename T>
inline T const& myMin (T const& a, T const& b)//模板函数实现任意数据类型的数据求最
小值
{
    return  a < b ? a : b;
}
bool isInt(string s)//根据输入数据是否小数点来判断是整形数还是浮点数
{
    stringstream sin(s);
    char c;
    while(sin>>c)
    {
        if(c == '.')
            return false;
    }
    return true;
}
```

```
int main(int argc, char *argv[])
{
    if (argc != 3)//参数个数不对
    {
        cout << "the number of parameters is error!" << endl;
        exit(-1);
    }
    string s1 = argv[1], s2 = argv[2];
    if(isInt(s1) && isInt(s2))//整形输入
    {
        cout<<"among "<<s1<<" and "<<s2<<", "<<"the smaller is:
"<<myMin(atoi(s1.data()),atoi(s2.data()))<<endl;
    }
    else//浮点型输入
    {
        cout<<"among "<<s1<<" and "<<s2<<", "<<"the smaller is:
"<<myMin(atof(s1.data()),atof(s2.data()))<<endl;
    }
    return 0;
}
```

- average(m,n,l)的实现：

```
/**
 *
 * @author:     胡文博
 * @email:      huwenbo@mail.dlut.edu.cn
 * @dateTime:       2017-05-15 19:32:46
 * @description:
 */
#include <cstdlib>
#include <string>
#include <iostream>
#include <sstream>
using namespace std;

template <typename T>
inline T  myAverage (T a, T b, T c)//模板函数实现任意数据类型的数据求平均值·
{
    return  (a + b + c)/3;
}
bool isInt(string s)//根据输入数据是否小数点来判断是整形数还是浮点数
{
    stringstream sin(s);
    char c;
    while (sin >> c)
    {
        if (c == '.')
            return false;
    }
    return true;
}

int main(int argc, char *argv[])
```

```
{
    if (argc != 4)//参数个数不对
    {
        cout << "the number of parameters is error, average func need 3
parameters!" << endl;
        exit(-1);
    }
    string s1 = argv[1], s2 = argv[2], s3 = argv[3];
    if (isInt(s1) && isInt(s2) && isInt(s3))//整形输入
    {
        // cout<<"h"<<endl;
        cout << "the average of  " << s1 << ", " << s2 << " and " << s3 << " is:
" << myAverage(atoi(s1.data()), atoi(s2.data()),atoi(s3.data())) << endl;
    }
    else//浮点型输入
    {
        cout << "the average of  " << s1 << ", " << s2 << " and " << s3 << " is:
" << myAverage(atof(s1.data()), atof(s2.data()),atof(s3.data())) << endl;
    }
    return 0;
}
```

- 命令处理程序 shell 的实现：

```
/**
 *
 * @author:     胡文博
 * @email:      huwenbo@mail.dlut.edu.cn
 * @dateTime:       2017-05-15 19:52:46
 * @description:
 */
#include <iostream>
#include <unistd.h>
#include <cstdlib>
#include <string>
#include <vector>
#include <boost/algorithm/string.hpp>
#include <boost/format.hpp>
#include <boost/tokenizer.hpp>
using namespace std;
using namespace boost::algorithm;

int main(int argc, char const *argv[])
{
    cout << "welcome to young shell !" << endl;
    string s;
    cout << "young@shel: ->_->";
    vector<string> vecStr;
    while (getline(cin, s))//读入一行输入
    {
        cout << "young@shel: ->_->";//打印命令提示符
        vecStr.clear();
        boost::char_separator<char> sep("  ,()");
        typedef boost::tokenizer<boost::char_separator<char> >
        CustonTokenizer;
```

```
        CustonTokenizer tok(s, sep);//正则表达式将输入字符串分割为命令和参数
        for (CustonTokenizer::iterator beg = tok.begin(); beg != tok.end();
++beg)
        {
            vecStr.push_back(*beg);
        }
        if (vecStr.size())//非空输入
        {
            if (fork() == 0)//判断是否位于子进程
            {
                if (vecStr[0] == "max")//子进程装入 max(m,n)程序数据
                {

execlp("/home/cris/gitRep/OS_Experiment/ProcessManange_ex1/build/max",
"./max", vecStr[1].data(), vecStr[2].data(), NULL);
                    exit(0);
                }
                else if (vecStr[0] == "min")//子进程装入 min(m,n)程序数据
                {

execlp("/home/cris/gitRep/OS_Experiment/ProcessManange_ex1/build/min",
"./min", vecStr[1].data(), vecStr[2].data(), NULL);
                    exit(0);
                }
                else if (vecStr[0] == "average")//子进程装入 average(m,n,1)程序数
据
                {

execlp("/home/cris/gitRep/OS_Experiment/ProcessManange_ex1/build/average",
"./average", vecStr[1].data(), vecStr[2].data(), vecStr[3].data(), NULL);
                    exit(0);
                }
                else
                    exit(0);
            }
        }
    }
    return 0;
}
```

1.3.3 实验结果

实验结果如图 1.5 所示，可见代码实现了实验要求。

2. 实验二：处理器调度

随机给出一个进程调度实例，如图 2.1 所示，模拟进程调度，给出按照算法先来先服务 FCFS、轮转 RR（ q=1）、最短进程优先 SJF、最高响应比优先 HRN 进行调度各进程的完成时间、周转时间、 带权周转时间。

进程	到达时间	服务时间
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

图 2.1

2.1 实验分析

本次实验采用 C++语言实现，采用了面向对象的编程思想，首先抽象出来两个类 process 和 processSchema，其中 process 类实现了对进程的描述，包含了进程的基本信息以及进程的各种动作，processSchema 类则实现了从文件读取题目中给定的进程调度实例，并将各进程存储在一个 vector 中，实现了记录所有进程的各种信息的功能。然后分别实现各个调度算法来调度 processSchema 中的进程。

在 FCFS 调度算法中，使用一个队列的数据结构来存储就绪进程队列，每到来一个新进程就将其放在队尾，每执行完一个进程进行一次调度，每次调度的内容是将队首的进程拿出并为其分配 CPU 进行执行，直至执行完毕。

在 RR 调度算法中，同样使用一个队列来存储就绪的进程队列，不同的是调度的时机，在 RR 调度算法中每当一个时间片用尽便会进行一次调度，调度过程是将当前正在执行的进程从队首拿出置于队尾，再给队首的进程分配 CPU 进行执行，并且每当新创建一个进程时便将该进程加入队尾等待调度，直至队列中所有进程执行完毕。

在 SJF 调度算法中，为了便于随机存取使用了 vector 来存储就绪的进程队列，调度的时机仍是当有进程执行完毕时才进行调度，每次调度时选取所要求的服务时间最短的进程进行执行，直至执行完毕所有进程。

在 HRN 调度算法中，使用 vector 存储就绪的进程，调度时机是进程执行完毕时，不同的是选择分配 CPU 的进程时依据的不再是静态的进程的服务时间，而是动态的等待时

间和需要服务的时间之和再除以需要服务时间作为调度的优先权，直至执行完毕所有进程。

2.2 代码实现

- process 类的实现

process.h:

```
/**
 *
 * @author:     胡文博
 * @email:      huwenbo@mail.dlut.edu.cn
 * @dateTime:       2017-06-04 15:28:14
 * @description:
 */
#ifndef PROCESS_H
#define PROCESS_H
#include<string>
using namespace std;
class process
{
    int ID;//进程 ID 号
    int workTime = 0;//已经执行的时间
public:
    int serviceTime,//进程需要的服务时间
        comeTime,//进程到来的时间
        finishTime;//进程执行结束的时间点
    string name;//进程名
    process();
    process(int ID, int serviceTime, int comeTime, string name);
    void run();//执行进程
    bool isFinished();//查询进程是否执行完毕
    void disp();//打印进程信息
    void dispResult();//打印执行结果
    int getTurnaroundTime();//计算周转时间
    double getWeightedTurnaroundTime();//计算带权周转时间
};

#endif // PROCESS_H
```

process.cpp:

```
/**
 *
 * @author:     胡文博
 * @email:      huwenbo@mail.dlut.edu.cn
 * @dateTime:       2017-06-04 15:27:50
 * @description:
 */
#include "process.h"
#include <iostream>
```

```
using namespace std;
process::process()
{
    this->workTime = 0;
    this->name = "process";
}
process::process(int ID, int serviceTime, int comeTime, string name = "process")
{
    this->ID = ID;
    this->serviceTime = serviceTime;
    this->comeTime = comeTime;
    this->name = name;
    this->workTime = 0;
    this->finishTime = 0;
}
void process::disp()//打印进程信息
{
    cout << "" << this->name << "     " << this->ID << "      " << this->comeTime
<< "          " << this->serviceTime << endl;
}
void process::dispResult()
{
    cout << "" << this->name << "     " << this->ID << "      " << this->finishTime
<< "\t\t\t\b\b\b\b" << this->getTurnaroundTime() <<"\t\t\t"<<
this->getWeightedTurnaroundTime()<<endl;
}
void process::run()//执行进程
{
    workTime++;
}
bool process::isFinished()//查询进程是否执行完毕
{
    if (workTime >= serviceTime)
    {
        return true;
    }
    return false;
}
int process::getTurnaroundTime()//计算周转时间
{
    return finishTime-comeTime;
}
double process::getWeightedTurnaroundTime()//计算带权周转时间
{
    return (double)getTurnaroundTime()/serviceTime;
}
```

processSchema.h:

```
/**
 *
 * @author:     胡文博
 * @email:      huwenbo@mail.dlut.edu.cn
 * @dateTime:       2017-06-04 15:28:33
```

```
 * @description:
 */
#ifndef PROCESSSCHMA_H
#define PROCESSSCHMA_H
#include"process.h"
#include <vector>
#include <string>
using namespace std;

class processSchma
{
    void readFromFile(string fileName);//从文件读取进程调度实例
public:
    vector<process> processVec;//存储进程
    processSchma();
    processSchma(string fileName);
    void disp();//打印所有进程信息
    void dispResult();//打印所有进程结果信息
};
#endif // PROCESSSCHMA_H
```

processSchema.cpp:

```
/**
 *
 * @author:     胡文博
 * @email:      huwenbo@mail.dlut.edu.cn
 * @dateTime:       2017-06-04 14:58:47
 * @description:
 */
#include "processschma.h"
#include <fstream>
#include <iostream>
using namespace std;
processSchma::processSchma()
{}
processSchma::processSchma(string fileName)
{
    readFromFile(fileName);
}
void processSchma::readFromFile(string fileName)//从文件读取进程调度实例
{
    fstream f;
    f.open(fileName);
    string s;
    getline(f,s);
    int ID = 1;
    while(f>>s)
    {
        string name = s;
        f>>s;
        int comeTime = atoi(s.data());
        f>>s;
        int serviceTime = atoi(s.data());
```

```
        process indexP(ID++,serviceTime,comeTime,name);
        this->processVec.push_back(indexP);
    }
}
void processSchma::disp()//打印所有进程信息
{
    cout<<"-----------------------------------------"<<endl
        <<"Name  ID  createTime  serviceTime"<<endl
        <<"-----------------------------------------"<<endl;
    for(vector<process>::iterator iter = processVec.begin(); iter !=
processVec.end(); ++iter)
    {
        iter->disp();
    }
    cout<<"-----------------------------------------"<<endl;
}
void processSchma::dispResult()//打印所有进程结果信息
{

cout<<"-------------------------------------------------------------------"<
<endl
        <<"Name  ID  finishTime  turnaroundTime  weightedTurnaroundTime"<<endl

<<"-------------------------------------------------------------------"<<end
l;
    for(vector<process>::iterator iter = processVec.begin(); iter !=
processVec.end(); ++iter)
    {
        iter->dispResult();
    }

cout<<"-------------------------------------------------------------------"<
<endl;
}
```

main.cpp:

```
/**
 *
 * @author:     胡文博
 * @email:      huwenbo@mail.dlut.edu.cn
 * @dateTime:       2017-06-03 21:18:17
 * @description:
 */
#include <iostream>
#include "process.h"
#include "processschma.h"
#include <queue>
#include <boost/circular_buffer.hpp>
#include <map>
using namespace std;
//检查是否有进程到来，进程采用 queue 存储
void checkComeProcess( queue<process*> &comeonProcess, processSchma& schema,
int time )
```

```
{
    for (vector<process>::iterator iter = schema.processVec.begin(); iter !=
schema.processVec.end(); iter++)
    {
        if ( iter->comeTime == time )
        {
            comeonProcess.push(&(*iter));
        }
    }
}
//重载 checkComeProcess 函数，区别是进程采用 vector 存储
void checkComeProcess( vector<process*> &comeonProcess, processSchma& schema,
int time )
{
    for (vector<process>::iterator iter = schema.processVec.begin(); iter !=
schema.processVec.end(); iter++)
    {
        if ( iter->comeTime == time )
        {
            comeonProcess.push_back( &(*iter));
        }
    }
}
//先来先服务算法的调度实现
void FCFS(processSchma& schema)
{
    int time = 0;
    queue<process*> comeonProcess;
    int processNum = schema.processVec.size();
    checkComeProcess(comeonProcess, schema, time);
    while (processNum)//进程还未执行完便继续
    {
        if (comeonProcess.empty())//执行队列为空
        {
            time ++;
            checkComeProcess(comeonProcess, schema, time);
            continue;
        }
        comeonProcess.front()->run();//执行队首程序
        time ++;
        checkComeProcess(comeonProcess, schema, time);
        if (comeonProcess.front()->isFinished())//队首程序执行完毕
        {
            comeonProcess.front()->finishTime = time;
            comeonProcess.pop();
            processNum --;
        }
    }
}
//时间片轮转法调度算法实现，时间片为 1
void RR(processSchma& schema)
{
    int time = 0;
    int processNum = schema.processVec.size();
```

```
    queue<process*> comeonProcess;
    checkComeProcess(comeonProcess, schema, time);
    while (processNum)//进程还未执行完便继续
    {
        if (comeonProcess.empty())//执行队列为空
        {
            time ++;
            checkComeProcess(comeonProcess, schema, time);
            continue;
        }
        auto tmp = comeonProcess.front();//执行队首程序
        comeonProcess.pop();
        tmp->run();
        time++;
        checkComeProcess(comeonProcess, schema, time);
        if (tmp->isFinished())//队首程序执行完毕
        {
            tmp->finishTime = time;
            processNum --;
        }
        else//将未执行完的队首程序置于队尾
            comeonProcess.push(tmp);
    }
}
//短进程优先调度算法实现
void SJF(processSchma& schema)
{
    int time = 0;
    vector<process*> comeonProcess;
    int processNum = schema.processVec.size();
    checkComeProcess(comeonProcess, schema, time);
    while (processNum)//进程还未执行完便继续
    {
        if (comeonProcess.empty())//执行队列为空
        {
            checkComeProcess(comeonProcess, schema, ++time);
            continue;
        }
        auto tmp = comeonProcess.begin();
        //寻找服务时间最短的进程
        for (auto index = tmp; index != comeonProcess.end(); index++)
        {
            if ((*index)->serviceTime < (*tmp)->serviceTime)
                tmp = index;
        }
        auto tmpProcess = *tmp;
        comeonProcess.erase(tmp);
        while (!tmpProcess->isFinished() )//直到该进程执行完毕
        {
            tmpProcess->run();
            checkComeProcess(comeonProcess, schema, ++time);
        }
        processNum--;
```

```
        tmpProcess->finishTime = time;
    }
}
//高响应比优先调度算法的实现
void HRN(processSchma& schema)
{
    int time = 0;
    vector<process*> comeonProcess;
    int processNum = schema.processVec.size();
    checkComeProcess(comeonProcess, schema, time);
    while (processNum)//进程还未执行完便继续
    {
        if (comeonProcess.empty())//执行队列为空
        {
            checkComeProcess(comeonProcess, schema, ++time);
            continue;
        }
        auto tmp = comeonProcess.begin();
        // 寻找优先级最高的进程
        for (auto index = tmp; index != comeonProcess.end(); index++)
        {
            if ( ((time - (*index)->comeTime) / (double)(*index)->serviceTime ) >
((time - (*tmp)->comeTime) / (double)(*tmp)->serviceTime ) )      //compare
waittingtime/servicetime
                tmp = index;
        }
        auto tmpProcess = *tmp;
        comeonProcess.erase(tmp);
        while (!tmpProcess->isFinished() )//直到该进程执行完毕
        {
            tmpProcess->run();
            checkComeProcess(comeonProcess, schema, ++time);
        }
        processNum--;
        tmpProcess->finishTime = time;
    }
}
// 主函数
int main(int argc, const char *argv[])
{
    if (argc != 3)//输入格式不对，并提醒
    {
        cout << "please use the format: ProcesserSchema_ex2 [process list file]
[schema algorithm]" << endl;
        return -1;
    }
    processSchma pS(argv[1]);
    cout << "the given process list is:" << endl;
    pS.disp();
    map<string, function<void(processSchma&)> > myEval;//建立字符串到函数的映射
    myEval["FCFS"] = FCFS;
    myEval["RR"] = RR;
    myEval["SJF"] = SJF;
    myEval["HRN"] = HRN;
```

```
    string comand(argv[2]);
    myEval[comand](pS);
    cout << "\n\n\nwork over, and the result is:" << endl;
    pS.dispResult();//打印结果
    return 0;
}
```

2.3 实验结果

实验结果如图 2.2 所示

3. 实验三：存储管理上级作业

1. 示例实验程序中模拟两种置换算法： LRU 算法和 FIFO 算法。
2. 给定任意序列不同的页面引用序列和任意分配页面数目，显示两种
算法的页置换过程。
3. 能统计和报告不同置换算法情况下依次淘汰的页号、缺页次数（页错误数）和缺页率。

3.1 实验分析

感　　想

附录 A　附录内容名称

以下内容可放在附录之内：

（1） 正文内过于冗长的公式推导；

（2） 方便他人阅读所需的辅助性数学工具或表格；

（3） 重复性数据和图表；

（4） 论文使用的主要符号的意义和单位；

（5） 程序说明和程序全文

（6） 调研报告。

这部分内容可省略。如果省略，删掉此页。

书写格式说明：

标题“附录 A 附录内容名称”选用模板中的样式所定义的“附录”；或者手动设置成字体：黑体，居中，字号：小三，1.5 倍行距，段后 1 行，段前为 0 行。

附录正文选用模板中的样式所定义的“正文”，每段落首行缩进 2 字；或者手动设置成每段落首行缩进 2 字，字体：宋体，字号：小四，行距：多倍行距 1.25，间距：段前、段后均为 0 行。