```c
/*
 * File: pi-sched.c
 * Author: Andy Sayler
 * Modifier: Chris Sterling
 * Project: CSCI 3753 Programming Assignment 3
 * Create Date: 2012/03/07
 * Modify Date: 2012/03/09
 * Description:
 *  This file contains a simple program for statistically
 *      calculating pi using a specific scheduling policy.
 */

/* Local Includes */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <errno.h>
#include <sched.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define DEFAULT_ITERATIONS 1000000
#define RADIUS (RAND_MAX / 2)

inline double dist(double x0, double y0, double x1, double y1){
    return sqrt(pow((x1-x0),2) + pow((y1-y0),2));
}

inline double zeroDist(double x, double y){
    return dist(0, 0, x, y);
}
void calculate_pi(long iterations)
{
    double x, y;
    long i;
    double inCircle = 0.0;
    double inSquare = 0.0;
    double pCircle = 0.0;
    double piCalc = 0.0;
    //BEGIN CPU BOUND ALGORITHM
        /* Calculate pi using statistical method across all iterations */
        for(i=0; i<iterations; i++)
        {
            x = (random() % (RADIUS * 2)) - RADIUS;
            y = (random() % (RADIUS * 2)) - RADIUS;
            if(zeroDist(x,y) < RADIUS)
            {
                inCircle++;
            }
            inSquare++;
        }
        /* Finish calculation */
        pCircle = inCircle/inSquare;
        piCalc = pCircle * 4.0;
        /* Print result */
        //fprintf(stdout, "pi = %f\n", piCalc);
    //END ALGORITHM
}

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

    long i;
    long iterations;
    struct sched_param param;
    int policy;
    pid_t pid;
    int nChildren;
    pid_t *pids;

    /* Process program arguments to select iterations and policy */
    /* Set default iterations if not supplied */
    if(argc < 2)
    {
        iterations = DEFAULT_ITERATIONS;
    }
    /* Set default policy if not supplied */
    if(argc < 3)
    {
        policy = SCHED_OTHER;
    }
    /*Set nChildren if not supplied*/
    if(argc < 4)
    {
        nChildren = 5;
    }
    /* Set iterations if supplied */
    if(argc > 1)
    {
        iterations = atol(argv[1]);
        if(iterations < 1)
        {
            fprintf(stderr, "Bad iterations value\n");
            exit(EXIT_FAILURE);
        }
    }
    /* Set policy if supplied */
    if(argc > 2)
    {
        if(!strcmp(argv[2], "SCHED_OTHER")){
            policy = SCHED_OTHER;
        }
        else if(!strcmp(argv[2], "SCHED_FIFO")){
            policy = SCHED_FIFO;
        }
        else if(!strcmp(argv[2], "SCHED_RR")){
            policy = SCHED_RR;
        }
        else{
            fprintf(stderr, "Unhandeled scheduling policy\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Set nChildren if supplied */
    if(argc > 3)
    {
        nChildren = atol(argv[3]);
        if(nChildren < 1)
        {
            fprintf(stderr, "Bad childrens value\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Set process to max priorty for given scheduler */
    param.sched_priority = sched_get_priority_max(policy);

    /* Set new scheduler policy */
    //fprintf(stdout, "Current Scheduling Policy: %d\n", sched_getscheduler(0));
    //fprintf(stdout, "Setting Scheduling Policy to: %d\n", policy);
    if(sched_setscheduler(0, policy, &param))
    {
        perror("Error setting scheduler policy");
        exit(EXIT_FAILURE);
    }
    //fprintf(stdout, "New Scheduling Policy: %d\n", sched_getscheduler(0));

    pids = malloc(nChildren * sizeof(pid_t)); //create an array to hold all our children


    for (i = 1; i <= nChildren; i++) {
        pids[i] = fork();
        if (pids[i] == -1)
        {
            return EXIT_FAILURE; //if a single one of our processes failed, fail the program
        }
        else if (pids[i] == 0)
        {
            //printf("I am a child: %d PID: %d\n",i, getpid());
            calculate_pi(iterations);
            exit(0); //when done with the pi calculation, exit
        }
        else
        {
            //I am the parent - I don't need to do anything in here

        }
    }
    // Wait for children to exit.
    int status;
    //when the loop starts, i = nChildren, so we can use i as our counter still
    while (i > 0)
    {
        pid = wait(&status);
        //printf("Child with PID %ld exited with status 0x%x.\n", (long)pid, status);
        --i;
    }
    free(pids);
    return EXIT_SUCCESS;
}
```