```c
/* 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>
#include <fcntl.h>
#include <sys/stat.h>

#define DEFAULT_ITERATIONS 1000000
#define RADIUS (RAND_MAX / 2)
#define MAXFILENAMELENGTH 80
#define DEFAULT_TRANSFERSIZE 1024*100

#define USAGE "./mixed <iterations> <policy> <children>"

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);
}
double 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);
```

```
        return piCalc;
    //END ALGORITHM
}
void log_pi(long iterations, int id)
{
    int outputFD;
    ssize_t bytesWritten;
    //ssize_t bytesRead;
    char transferBuffer[DEFAULT_TRANSFERSIZE];
    char outputFilename[MAXFILENAMELENGTH];
    long i;
    //for(i=0;i<iterations;i++)
    //{
        snprintf(outputFilename, MAXFILENAMELENGTH, "%s-%d", "pilog", id); //create the output
        filename
        if((outputFD = open(outputFilename,
             O_WRONLY | O_CREAT | O_TRUNC | O_SYNC,
             S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH)) < 0) //open the output file for
             this process
        {
            perror("Failed to open output file");
            exit(EXIT_FAILURE);
        }
        //ltoa(calculate_pi(iterations), transferBuffer, 1);
        sprintf(transferBuffer, "%ld", calculate_pi(iterations));
        bytesWritten = write(outputFD, transferBuffer, DEFAULT_TRANSFERSIZE); //write out the
        pi results to the output file
        if(close(outputFD))
        {
            perror("Failed to close input file");
            exit(EXIT_FAILURE);
        }
    //}
}

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;
    }
    else
    {
        iterations = atol(argv[1]);
```

```
        if(iterations < 1)
        {
            fprintf(stderr, "Bad iterations value\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Set default policy if not supplied */
    if(argc < 3)
    {
        policy = SCHED_OTHER;
    }
    else
    {
        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 not supplied*/
    if(argc < 4)
    {
        nChildren = 5;
    }
    else
    {
        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 */
```

```
    if(sched_setscheduler(0, policy, &param))
    {
        perror("Error setting scheduler policy");
        exit(EXIT_FAILURE);
    }

    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());
            log_pi(iterations, i);
            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;
}
```