```
/*
 * File: rw.c
 * Author: Andy Sayler
 * Modifier: Chris Sterling
 * Project: CSCI 3753 Programming Assignment 3
 * Create Date: 2012/03/19
 * Modify Date: 2012/03/20
 * Description: A small i/o bound program to copy N bytes from an input
 *              file to an output file. May read the input file multiple
 *              times if N is larger than the size of the input file.
 */

/* Include Flags */
#define _GNU_SOURCE

/* System Includes */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include <sched.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <pthread.h>

/* Local Defines */
#define MAXFILENAMELENGTH 80
#define DEFAULT_INPUTFILENAME "rwinput"
#define DEFAULT_OUTPUTFILENAMEBASE "rwoutput"
#define DEFAULT_BLOCKSIZE 1024
#define DEFAULT_TRANSFERSIZE 1024*100

#define USAGE "./rw <transfersize> <blocksize> <inputfile> <outputfile> <policy> <nChildren>"

void do_rw(ssize_t transfersize, ssize_t blocksize, char* inputFilenameBase, char*
outputFilenameBase, char* transferBuffer, ssize_t buffersize, int id)
{

    ssize_t bytesRead = 0;
    ssize_t totalBytesRead = 0;
    int totalReads = 0;
    ssize_t bytesWritten = 0;
    ssize_t totalBytesWritten = 0;
    int totalWrites = 0;
    int inputFileResets = 0;
    int rv;
    int inputFD;
    int outputFD;
    char outputFilename[MAXFILENAMELENGTH];
    char inputFilename[MAXFILENAMELENGTH];
```

```
    /* Open Input File Descriptor in Read Only mode */
    snprintf(inputFilename, MAXFILENAMELENGTH, "%s-%d", inputFilenameBase, id);
    if((inputFD = open(inputFilename, O_RDONLY | O_SYNC)) < 0)
    {
        fprintf(stderr, "Failed to open input file %s\n", inputFilename);
        exit(EXIT_FAILURE);
    }

    /* Open Output File Descriptor in Write Only mode with standard permissions*/
    rv = snprintf(outputFilename, MAXFILENAMELENGTH, "%s-%d", outputFilenameBase, getpid());
    if(rv > MAXFILENAMELENGTH)
    {
        fprintf(stderr, "Output filename length exceeds limit of %d characters.\n",
        MAXFILENAMELENGTH);
        exit(EXIT_FAILURE);
    }
    else if(rv < 0)
    {
        perror("Failed to generate output filename");
        exit(EXIT_FAILURE);
    }
    if((outputFD =
    open(outputFilename,
         O_WRONLY | O_CREAT | O_TRUNC | O_SYNC,
         S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH)) < 0)
    {
        perror("Failed to open output file");
        exit(EXIT_FAILURE);
    }

    /* Print Status */
    //fprintf(stdout, "Reading from %s and writing to %s\n", inputFilename, outputFilename);

    /* Read from input file and write to output file*/
    do{
        /* Read transfersize bytes from input file*/
        bytesRead = read(inputFD, transferBuffer, buffersize);
        if(bytesRead < 0)
        {
            perror("Error reading input file");
            exit(EXIT_FAILURE);
        }
        else
        {
            totalBytesRead += bytesRead;
            totalReads++;
        }

        /* If all bytes were read, write to output file*/
        if(bytesRead == blocksize)
        {
            bytesWritten = write(outputFD, transferBuffer, bytesRead);
            if(bytesWritten < 0)
            {
```

```
                perror("Error writing output file");
                exit(EXIT_FAILURE);
            }
            else
            {
                totalBytesWritten += bytesWritten;
                totalWrites++;
            }
        }
        /* Otherwise assume we have reached the end of the input file and reset */
        else
        {
            if(lseek(inputFD, 0, SEEK_SET))
            {
                perror("Error resetting to beginning of file");
                exit(EXIT_FAILURE);
            }
            inputFileResets++;
        }
    }while(totalBytesWritten < transfersize);

    /* Output some possibly helpfull info to make it seem like we were doing stuff */
    //fprintf(stdout, "Read:    %zd bytes in %d reads\n", totalBytesRead, totalReads);
    //fprintf(stdout, "Written: %zd bytes in %d writes\n", totalBytesWritten, totalWrites);
    //fprintf(stdout, "Read input file in %d pass%s\n", (inputFileResets + 1), (inputFileResets
    ? "es" : ""));
    //fprintf(stdout, "Processed %zd bytes in blocks of %zd bytes\n", transfersize, blocksize);



    /* Close Output File Descriptor */
    if(close(outputFD))
    {
        perror("Failed to close output file");
        exit(EXIT_FAILURE);
    }

    /* Close Input File Descriptor */
    if(close(inputFD))
    {
        perror("Failed to close input file");
        exit(EXIT_FAILURE);
    }

}
int main(int argc, char* argv[])
{
    ssize_t transfersize = 0;
    ssize_t blocksize = 0;
    ssize_t buffersize;
    pid_t pid;
    pid_t *pids;

    int nChildren;
```

```
    int policy;
    long i;

    char inputFilenameBase[MAXFILENAMELENGTH];
    char outputFilenameBase[MAXFILENAMELENGTH];
    char* transferBuffer = NULL;

    struct sched_param param;

    /* Process program arguments to select run-time parameters */
    /* Set supplied transfer size or default if not supplied */
    if(argc < 2)
    {
        transfersize = DEFAULT_TRANSFERSIZE;
    }
    else
    {
        transfersize = atol(argv[1]);
        if(transfersize < 1)
        {
            fprintf(stderr, "Bad transfersize value\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Set supplied block size or default if not supplied */
    if(argc < 3)
    {
        blocksize = DEFAULT_BLOCKSIZE;
    }
    else
    {
        blocksize = atol(argv[2]);
        if(blocksize < 1)
        {
            fprintf(stderr, "Bad blocksize value\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Set supplied input filename or default if not supplied */
    if(argc < 4)
    {
        if(strnlen(DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH) >= MAXFILENAMELENGTH)
        {
            fprintf(stderr, "Default input filename too long\n");
            exit(EXIT_FAILURE);
        }
        strncpy(inputFilenameBase, DEFAULT_INPUTFILENAME, MAXFILENAMELENGTH);
    }
    else
    {
        if(strnlen(argv[3], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH)
        {
```

```
            fprintf(stderr, "Input filename too long\n");
            exit(EXIT_FAILURE);
        }
        strncpy(inputFilenameBase, argv[3], MAXFILENAMELENGTH);
    }
    /* Set supplied output filename base or default if not supplied */
    if(argc < 5)
    {
        if(strnlen(DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH) >= MAXFILENAMELENGTH)
        {
            fprintf(stderr, "Default output filename base too long\n");
            exit(EXIT_FAILURE);
        }
        strncpy(outputFilenameBase, DEFAULT_OUTPUTFILENAMEBASE, MAXFILENAMELENGTH);
    }
    else
    {
        if(strnlen(argv[4], MAXFILENAMELENGTH) >= MAXFILENAMELENGTH)
        {
            fprintf(stderr, "Output filename base is too long\n");
            exit(EXIT_FAILURE);
        }
        strncpy(outputFilenameBase, argv[4], MAXFILENAMELENGTH);
    }
     /* Set policy if supplied */
    if(argc < 6)
    {
        policy = SCHED_OTHER;
    }
    else
    {
        if(!strcmp(argv[5], "SCHED_OTHER"))
        {
            policy = SCHED_OTHER;
        }
        else if(!strcmp(argv[5], "SCHED_FIFO"))
        {
            policy = SCHED_FIFO;
        }
        else if(!strcmp(argv[5], "SCHED_RR"))
        {
            policy = SCHED_RR;
        }
        else
        {
            fprintf(stderr, "Unhandeled scheduling policy\n");
            exit(EXIT_FAILURE);
        }
    }


    /* Set value for nChildren if supplied */
    if(argc < 7)
    {
```

```
        nChildren=5; //if no value supplied, set it to 5
    }
    else
    {
        nChildren = atol(argv[6]);
        if(nChildren < 1)
        {
            fprintf(stderr, "Bad childrens value\n");
            exit(EXIT_FAILURE);
        }
    }

    /* Confirm blocksize is multiple of and less than transfersize*/
    if(blocksize > transfersize)
    {
        fprintf(stderr, "blocksize can not exceed transfersize\n");
        exit(EXIT_FAILURE);
    }
    if(transfersize % blocksize)
    {
        fprintf(stderr, "blocksize must be multiple of transfersize\n");
        exit(EXIT_FAILURE);
    }

    /* Allocate buffer space */
    buffersize = blocksize;
    if(!(transferBuffer = malloc(buffersize*sizeof(*transferBuffer))))
    {
        perror("Failed to allocate transfer buffer");
        exit(EXIT_FAILURE);
    }

     /* Set process to max priority 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);
    }

    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());
            //ssize_t transfersize, ssize_t blocksize, char* inputFilename, char*
            outputFilename, char* transferBuffer, ssize_t buffersize
            do_rw(transfersize, blocksize, inputFilenameBase, outputFilenameBase, transferBuffer
            , buffersize, 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(transferBuffer);
    free(pids);
    return EXIT_SUCCESS;
}
```