Wollongong of University

School of Computer Science and Software Engineering

CSCI212 Interacting Systems Autumn 2015

Assignment 2 (7.5 marks)

Due 11:59pm Sunday April 26, 2014.

Aim

The main aim of this assignment is to become more familiar with the conceptual principals of scheduling. We will do this by writing a program to compute the average turnaround and waiting time for various algorithms. In addition to this we will become aware of how to use a library and how to build our own.

Task 1 (5 marks)

Step 1

In this assignment you are to write a program, which prompts you for the scheduler algorithm type and then reads in the details of processes till `EOF` (End of File).

When the program starts up it should query the user as to what algorithm they wish to use – selections include `FCFS` (First Come First Served), `SJF` (Shortest Job First), `SRT` (Shortest Remaining Time) and `RR` (Round Robin). The initial input should be as follows:

```
Welcome to Job Scheduler Computer
What algorithm would you like to use:
> RR
```

If the user enters `RR` the program should prompt for the size of the time quanta between context switches. The unit of measure used in this assignment is *micro seconds*.

```
What is the time quanta for RR scheduling:
> 100
```

Only the RR algorithm is to prompt for the time quanta. Other algorithm options include `FCFS`, `SJF` and `SRT`. Once the algorithm and quanta are provided then the user is prompted for the process details. The program will ask for process information till `EOF`. For each process you are to enter the burst time. For now you can assume that each process arrives at `time = 0`.

```
Process 1 (P1) – What is the burst time?
> 16
Process 2 (P2) – What is the burst time?
```

```
> 7
Process 3 (P3) – What is the burst time?
> 3
Process 4 (P4) – What is the burst time?
> 2
Process 5 (P5) – What is the burst time?
> EOF
```

Please note EOF does not denote the string `EOF` – rather it denotes end of stream is represented by Control `D` (`^D`).

Once the input has been entered, the program is to show the scheduling graph as illustrated in the subject lectures for the given algorithm. **The scheduling graph shows all periods in which the process is scheduled.**

For example the output of the above if the algorithm was FCFS would be:

```
Scheduled Jobs

Process    Burst Time          Start Time          Stop Time
P1         16                  0                16
P2         7                   16               23
P3         3                   23               26
P4         2                   26               28

Computing Wait times

P1         0
P2         16
P3         23
P4         26

Computing Turn around times

P1         16
P2         23
P3         26
P4         28
```

Notice how the wait and turnaround times are computed from `T = 0`. In addition to this you program should display the average waiting time for the processes and the average turnaround time.

Your job is to write a code solution that solves this problem in C. Place all your source code in the source file `sched1.c.`

Please note in this task SJF and SRT is effectively the same algorithm because the arrival time of the processes is the same. In the next section we draw the distinction through pre-emption.

Step 2

The code you implemented for task 1 assumes that the processes all arrive at the same time i.e. T = 0. As a result algorithms such as SJF and SRT behave the same.

Lets now add to our program, so that processes can arrive at any time.

In order for this to work reasonably well, you should make it so the start time of the next process is no greater then the cumulative time of all the previous processes.

Additionally you can assume the start time of the process is no smaller then its predecessors. If either of these constraints is not met an error should be printed and displayed.

For each piece of process information you gather, you will now need to get the start time of the process.

For example:

```
Welcome to Job Scheduler Computer
What algorithm would you like to use:
> FCFS
Process 1 (P1) – What is the burst time & start time?
> 16 0
Process 2 (P2) – What is the burst time & start time?
> 7 2
Process 3 (P3) – What is the burst time & start time?
> EOF
```

In the above case process 1 arrives at T = 0 and Process 2 arrives at T=2. If the start time for process 2 was greater then 16 then a suitable error should be printed.

The output from the above input would be.

```
Process   Burst Time          Start Time          Stop Time
P1        16                  0                   16
P2        7                   16                  23
```

Computing Wait times

```
P1        0
P2        14
```

Computing Turn around times

```
P1        16
P2        21
```

Again your program should also print out the average turnaround and average wait time. You are implementing this style of behaviour for all algorithms.

You can assume for this part the SJF is non-pre-emptive where as SRT is a pre-emptive version of SJF. This therefore means when using SRT that if a process arrives during the CPU burst of a executing process, with a smaller burst time, the current process will be scheduled out and the process with the smaller burst time will be scheduled in. You are to place your solution for this task in the source file `sched2.c`.

Step 3

Your next step is to make a few changes to sched1.c and sched2.c. Firstly make it so that both programs can not take more then 64 processes as data entry.

Once you have done that the next step is to take the data you have in text and present it graphically. You will still output the summaries as above to standard output but before asking for the scheduler type you will ask for an output file name – this is the file an image will be generated to.

```
Welcome to Job Scheduler Computer
Enter a filename for the output image:
> foo.png
What algorithm would you like to use:
```

To create the image we are you to use libGD which is a graphics library. LibGD can create images in a variety of formats including PNG. All images created will be of type PNG.

The library and headers can be found on banshee/ wumpus at `/packages/gd/`.

Your job is to do as follows:

1. Create a PNG Image representing the process scheduling
2. The default size of the image is 1024 x 768 – the graph should be centred
3. You are to allocate each process a color from a palette – you can allocate your own palette. *Explore some creativitiy*.
4. When plotting the time it is to be done in white, with notches representing process times as per example. The axis should be labelled as per example.
5. You are to show how all processes are interleaved. Each millisecond should be roughly 20 – 50 (its up to you) pixels wide. If the summation of time of your time exceeds the pixel count on the horizontal axis, double the images horizontal pixel count by 2 till the graph fits with some padding on either side.
6. Your graph is to have a key, the key is to be 4 elements high as per example and a maximum of 16 elements wide.

Add the code required to perform the above to `sched1.c` and `sched2.c`. For the time being the code can be in these files. You will need to write a suitable Makefile to compile these to targets `sched1` and `sched2`. The makefile should be called `Makefile`.

You will need to do some research as to the power of libgd. Read the header files, remember you must also use FILE*'s for I/O.

Step 4

Both `sched1.c` and `sched2.c` have many components in common, these being the scheduler itself and the plotting/drawing subsystem.

Your job now is to write a library, which has entry, points which perform the scheduling for both `sched1.c` and `sched2.c`. To do this you need to make your code generic. Place all code for the scheduler in a file called `libsched.c` and `libsched.h`.

In addition to this you are to extract the code for the graph and write a library as well – code should be placed in `libgrapher.cpp` and `libgrapher.h`.

You will now need to modify your makefile so that it produces a static library (.a) for your scheduler. The library should be called `libsched.a` and linked as `–lsched`. The makefile should also build the drawing library as `libgrapher.a` – that is a static library, linkage is to be done as `–lgrapher`.

You are to make appropriate changes to `sched1.c` and `sched2.c` so that they invoke your libraries. The Makefile should also be changed to link against the newly created libraries.

You can use the gcc compiler for this assignment.

Task 2 (2.5 marks)

For this part of the assignment you have to write a program which interprets ELF files and mimics the functionality of `ldd(1).` A typically invocation of `ldd(1)` is as follows;

```
$ ldd /usr/bin/ls
     libc.so.1 =>     /usr/lib/libc.so.1
     libdl.so.1 =>    /usr/lib/libdl.so.1
     /usr/platform/SUNW,Ultra-4/lib/libc_psr.so.1
```

`ldd(1)` tells you what dynamic libraries (shared objects) a program (the argument) depends upon. It does this by looking at the structure of the executable.

In Moodle there is a file called `elf.cpp`. When you inspect the `elf.cpp` you will notice it takes an argument. The program provides a list of all the libraries the argument depends on (if there are any). The program also prints out the additional search paths for libraries of the program.

If the arguement which is an executable exists it does the following;

1. Open the file using the statement `fd = open(argv[1], O_RDONLY))`
2. Set up the ELF library by associating it with the file `elf_begin(fd, ELF_C_READ, NULL)`
3. Scan through the ELF sections till the `.dynamic` section is found

```
while ((scn = elf_nextscn(elf, scn)) != NULL)
{
     gelf_getshdr(scn, &shdr);
                if (shdr.sh_type == SHT_DYNAMIC)
```

4. Once the dynamic section has been found print out the libraries that the object depends on by looking at element of type `DT_NEEDED` and the search path by looking at elements of type `DT_RPATH`. The search path is in `:` deli metered form and may contain many search paths.

To compile this simple little program simply enter the following;

```
g++ elf.cpp –o elf –lelf
```

The `–lelf` directive causes the compiler to link the program against the elf library which is normally in `/usr/lib`.

You job is to take `elf.cpp` and modify it so that it produce output similar to `ldd(1).` For example if you run `elf` with the following argument you should get.

```
$ elf /usr/bin/ls
Depends on Library: libc.so.1
```

Your job is to convert it to:

```
$ elf /usr/bin/ls
libc.so.1 =>     /usr/lib/libc.so.1
```

Another example:

```
$ elf /usr/bin/sh
Depends on Library: libgen.so.1
Depends on Library: libsecdb.so.1
Depends on Library: libnsl.so.1
Depends on Library: libc.so.1
```

Should be converted to:

```
$ elf /usr/bin/sh
libgen.so.1 =>   /usr/lib/libgen.so.1
libsecdb.so.1 =>         /usr/lib/libsecdb.so.1
libnsl.so.1 =>   /usr/lib/libnsl.so.1
libc.so.1 =>     /usr/lib/libc.so.1
```

The output that my `elf` program produces may differ slightly to `ldd(1)` – this is due to implementation of `elf.cpp`. How do you resolve the libraries? This is easy.

You can download the source for the library libfschecker. The library encompases a header file `fschecker.h`. If you inspect this file, you will notice there is one function.

```
bool checkfile(char* filename);
```

The function takes a fully qualified path and returns a Boolean to indicate if the file exists and is accessible to you. Compile the library and link it to your program. You can assume the library and header are in the same directory as your project.

To link this library into you program you should go;

```
$ g++ –I.
      -L.
       elf.cpp –o elf –lelf –lfschecker
```

To use the functionality from `libfschecker` you will need to include the header file in `elf.cpp`. The above compile directive will treat it as if it were in the systems header path.

To produce the output above, for each shared object dependency you identify simply invoke the function `checkfile(char* filename)` to validate its existence. But how do you know where to look for the library?

For the purpose of the assignment you should do the following;

1. Using the `:` delimited path provide by `DT_RPATH` in `elf.cpp` check to see if the library exists in any of those directories, if so print out the absolute path.
2. If no match check for the environment Variable `LD_LIBRARY_PATH.` If a match found print it out.
3. If no matches are found from `DT_PATH` then check for the library in `/usr/lib`. If a match found print it out.
4. If no matches are found then print out an error message.

To illustrate this consider the following example using `elf.cpp:`

```
$ elf /share/cs-pub/212/bin/sqlite3
Depends on Library: libsqlite3.so.0
Depends on Library: libcurses.so.1
Depends on Library: librt.so.1
Depends on Library: libc.so.1
Run time linkage path:
/share/cs-pub/212/lib
```

In this example, you will notice the runtime linkage path. There two of them. To resolve the dependencies simply take each library e.g. `libsqlite3.so.0` and check to see if it exists in any of the runtime linkage. If it did not exist in these location then check `/usr/lib`. Repeat the process for all other library dependencies.

```
$ elf /share/cs-pub/212/bin/sqlite3
libsqlite3.so.0 =>       /share/cs-pub/212/lib/libsqlite3.so.0
libcurses.so.1 =>        /usr/lib/libcurses.so.1
librt.so.1 =>    /usr/lib/librt.so.1
libc.so.1 =>     /usr/lib/libc.so.1
```

Again the output of `elf` differs from `ldd(1).` A more comprehensive test would be:

```
$ elf /packages/openldap/2.0.23/libexec/slapd
Depends on Library: libdb.so
Depends on Library: libssl.so.0
Depends on Library: libcrypto.so.0
```

```
Depends on Library: libresolv.so.2
Depends on Library: libgen.so.1
Depends on Library: libnsl.so.1
Depends on Library: libsocket.so.1
Depends on Library: libdl.so.1
Depends on Library: libpthread.so.1
Depends on Library: librt.so.1
Depends on Library: libc.so.1
Run time linkage path:
/packages/sleepycat/3/lib:/lib:/packages/openssl/0.9.6/lib
```

Once you are done though you should get:

```
$ elf packages/openldap/2.0.23/libexec/slapd
libdb.so =>      /packages/sleepycat/3/lib/libdb.so
libssl.so.0 =>   /packages/openssl/0.9.6/lib/libssl.so.0
libcrypto.so.0 =>        /packages/openssl/0.9.6/lib/libcrypto.so.0
libresolv.so.2 =>        /lib/libresolv.so.2
libgen.so.1 =>   /lib/libgen.so.1
libnsl.so.1 =>   /lib/libnsl.so.1
libsocket.so.1 =>        /lib/libsocket.so.1
libdl.so.1 =>    /lib/libdl.so.1
libpthread.so.1 =>       /lib/libpthread.so.1
librt.so.1 =>    /lib/librt.so.1
libc.so.1 =>     /lib/libc.so.1
```

Make suitable changes to `elf.cpp` to allow this to happen (you should ideally clean up the code into functions – note the C mechanics for function calls). You MUST use the compile directive stated above.

Feel free to modify the source code in `elf.cpp` to ensure you can get this task done.

You are to submit your modified `elf.cpp` for this part of the assignment. You should not introduce any additional files. Your code should compile with G++.

Submit:

Submit the final source files `sched1.cpp` and `sched2.cpp` along with the source for the libraries using the submit program. The directive is as follows;

```
submit -c csci212 -a ass2 sched1.c sched2.c libsched.c
          libsched.h libgrapher.c libgrapher.h Makefile elf.cpp
```

PLEASE NOTE – DUE TO THE DIFFERENCES OF LIBRARIES ON WRAITH CAN ASSUME THE LIBRARIES AND HEADERS IE LIBGB ARE IN YOUR CURRENT HOME DIRECTORY

An extension of time for the completion of the assignment may be granted in certain circumstances. A request for an extension must be made to the Subject Coordinator before the due date. Late assignments without granted extension will be marked but the mark awarded will be reduced by 1 mark for each day late. Assignments will not be accepted more than three days late.