University of Toronto Scarborough

CSCB09 Summer 2019 Final Examination

Duration - 2 hour 30 minutes

Last Name: Student Number:
First Name: UTORid:

1:	/10
2:	/16
3:	/10
4:	/24
5:	/20
Total:	/80

There are technical reminders after the questions.
There are blank pages for scratch work at the end.

Blank page for sketch work.

1. (a) [5 marks] Prof X has collected and organized assignment files from students into two levels of directories: First by assignments, then by students, e.g.,

```
Assignment 2/cliffmary
Assignment 2/jacksonjulia
Assignment 2/manningoscar
Lab 4/cliffmary
Lab 4/manningoscar
```

These are under a directory whose name is in the shell variable *src*. However, this *src* directory and the assignment directories may contain non-directory files for Prof X's self-reminders; these files should be skipped over in your shell script later.

Prof Y prefers a flatter directory structure: Just one level, each name contains the student name and the assignment name, separated by a space, a hyphen, a space:

```
cliffmary - Assignment 2
cliffmary - Lab 4
jacksonjulia - Assignment 2
manningoscar - Assignment 2
manningoscar - Lab 4
```

These should go under a directory whose name is in the shell variable *dest*.

And in the interest of sharing and saving space, Prof Y wants these to be symbolic links to Prof X's directories, as opposed to copying.

Implement a Bourne shell script for this. You may assume that there are really two levels of directories under *src*. You may assume that the *dest* directory exists and is empty initially. You should make minimum assumptions about directory/file names.

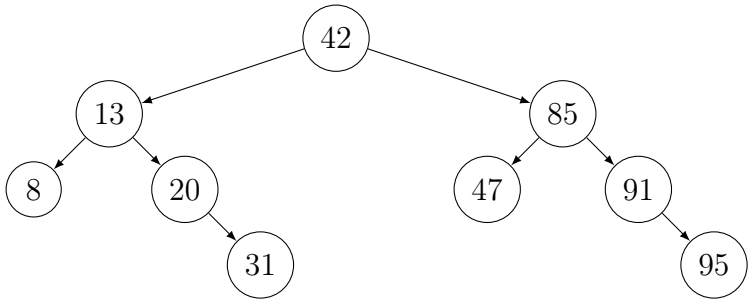
(b) [5 marks] Every B09 assignment that involves the 'fork' system call carries the risk (nay, *certainty*) of overloading the mathlab server because bugs may cause runaway processes (which is forgivable) and some students don't manually terminating their runaway processes (which is the unforgivable part). Prof A vows to deter the latter by deducting 1% from the final marks of the offending students for every 5 minutes of CPU time! (Note: If you process blocks at a syscall, it doesn't use CPU time, and it doesn't slow down mathlab.)

Prof A has already crafted a command to list processes, CPU times, and owners. Its output goes like this:

```
UID        PID         TIME
knuthd   27182     00:03:14
cliffm   31337     00:00:02
cliffm   31339     00:00:01
knuthd   31415    157:07:36
manning  44100     00:00:00
```

In reality the list contains hundreds of lines, so Prof A needs your help sorting it out. Write a Bourne shell pipeline that starts with Prof A's command (you may refer to it as 'pa-ps') but followed by your commands to output in decreasing order of CPU time, so Prof A can easily pick out the worst offenders. The header "UID PID TIME" should also be dropped. It is OK if the time format in the output becomes slightly different.

2. A binary search tree is a binary tree in which each node holds a key and a value, and if you perform an in-order traversal, you find the keys in strictly increasing order. For simplicity, in this question, the keys are `int`s and the values are omitted.



We use this C struct to represent a node:

```
typedef struct node {
  int key;
  struct node *left;   // pointer to left child node
  struct node *right;  // pointer to right child node
} node;
```

Note that either or both child pointers can be `NULL` to represent the absence of the respective child(ren). Functions for binary search trees also work with a pointer to a root node as a parameter and/or return value, and it too can be `NULL` to represent the empty tree.

(a) [4 marks] Implement a C function to print the keys in a given tree in decreasing order. Print one key per line.

```
void decreasing(const node *root)
{
```

```
void printInOrder(noBinTree *n){
   if(n != NULL){
      printInOrder(n->right);
      printf(" %d ", n->number);
      printInOrder(n->left);
   }
}
```

(b) [5 marks] Implement a C function to deallocate all nodes of the given tree, assuming that all nodes were dynamically allocated.

```
void free_tree(node *root)
{
```

```
void deleteTree(struct node* node)
{
   if (node == NULL) return;

   /* first delete both subtrees */
   deleteTree(node->left);
   deleteTree(node->right);

   /* then delete the node */
   printf("\n Deleting node: %d", node->data)
   free(node);
}
```

(c) [7 marks] An array *arr* of keys in strictly increasing order is given, along with two indexes i and j; i is a valid index for the array, j is at most the length of the array, and $i \leq j$. Implement a C function to build a binary search tree with the keys in the array from index i (inclusive) to index j (exclusive). Nodes are to be dynamically allocated (you may omit error checking). To produce a well-balanced tree, use this strategy:

- Use the median (key at middle) for the new root. To break ties, use the lower median, e.g., if $i = 1$ and $j = 5$, choose $arr[2]$ rather than $arr[3]$.
- The base cases are $i = j$ or $i + 1 = j$.

For example, if the array is { 3, 8, 13, 20, 31, 42, 47, 85, 91, 95, 97 } and $i = 1$, $j = 10$, then the built tree should look like the example tree at the beginning.

```
node *fromArray(const int *arr, int i, int j)
{
```

3. Olivia wishes to modularize the following C file into multiple C files for separate compilation.

```
typedef struct circle {
  double radius;
  double cx, cy;
} circle;

double circle_area(const circle *c) {
  return 3.14159 * c->radius * c->radius;
}

double cyl_vol(double radius, double height) {
  circle c;
  c.radius = radius;
  c.cx = 0; c.cy = 0;
  return area(&c) * height;
}

int main(void)
{
  printf("%f\n", cyl_vol(4, 10);
  return 0;
}
```

Olivia wishes to put `circle_area` in 'circle.c', `cyl_vol` in 'cyl.c', and `main` in 'myprog.c'; the `circle` struct also needs to be moved to a suitable header file.

(continued on next page)

Text

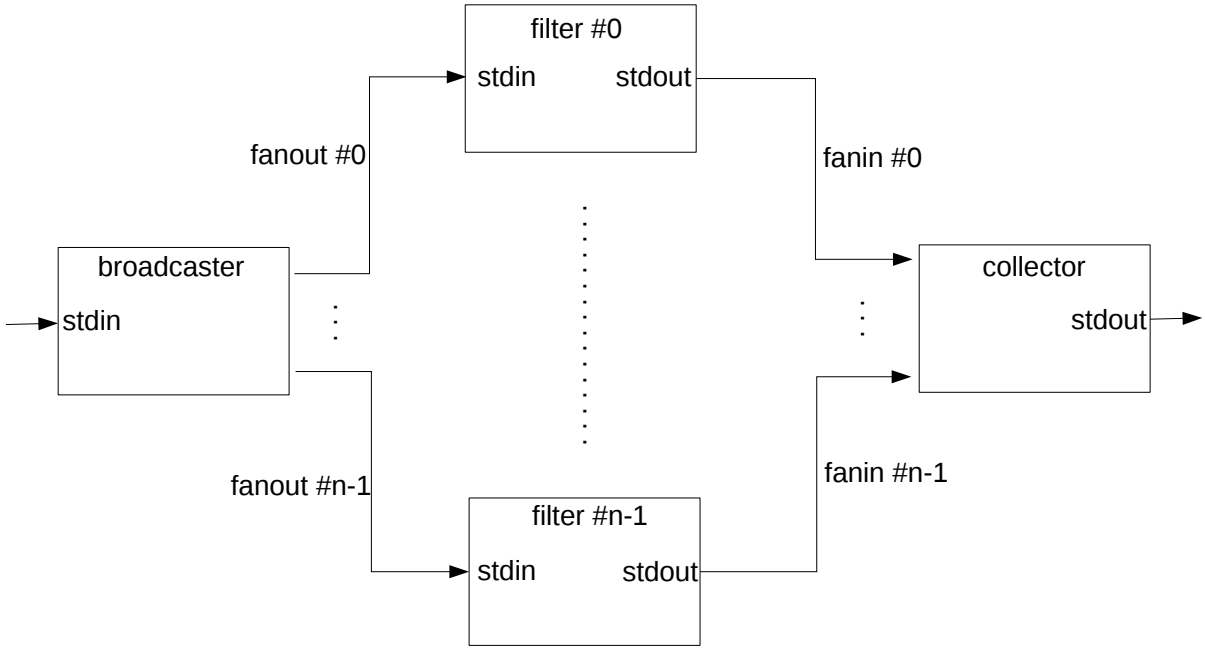
(a) [5 marks] Write the corresponding header files ‘circle.h’ and ‘cyl.h’.

(b) [5 marks] Write a Makefile for building an executable from the C source files after the modularization. The executable is to be called ‘myprog’.

Reminder for Makefile rule syntax:
target : *prerequisite*...
    command to build target

Reminder for ‘gcc’ options:
-c: compile to object files, e.g., ‘foo.c’ to ‘foo.o’
-o *name*: specify output filename

4. In this question, you will implement an ensemble of processes and pipes like in this diagram:



There are n filter processes. Each reads a sequence of real numbers from stdin (which comes from a fanout pipe, but the program doesn't need to know), and outputs a sequence of real numbers to stdout (which goes to a fanin pipe, but the program doesn't need to know). All filter processes run the same program code, but they receive different command line arguments—like biquad from Assignment 3, but just 1 argument instead of 5 for simplicity.

The broadcaster process copies data from its stdin to all n fanout pipes, so that all filter processes receive the same data. But the broadcaster should not care about the data format; it's just a copier.

The collector process sums the sequences from the fanin pipes termwise, and outputs the summed sequence. In other words, it reads one real number from each fanin pipe, adds them, outputs the sum to stdout; repeat.

All of these $n + 2$ processes have a common parent process (not shown).

The parent process, the broadcaster process, and the collector process run code from the same program—they just run different parts of the program code. This program is structured like:

```
int main(int argc, char **argv)
{
  // Mostly pseudocode shown.

  variable declarations and setup---omitted, but see below
```

```
  pipe creation, part (a)

  fork
  if child {
    broadcaster code, part (b)
  }

  fork
  if child {
    collector code, part (c)
  }

  fork-exec filters, part (d)

  wait for all children---omitted
  return 0;
}
```

The following variables have been declared and set as described in the comments.

```
int n;               // number of filter processes needed
char *filtername;    // pathname of filter program
char *filterarg[n];  // filterarg[i] is the command line argument
                     // for filter #i
```

The following variables have been declared, but you will have to set them:

```
int fanout[n][2], fanin[n][2];  // file descriptors of pipes
```

You may assume that I/O operations and syscalls have no error or failure apart from end of input.

(a) [2 marks] Write C code to create the $2n$ pipes and store their file descriptors in `fanout` and `fanin`.

(b) [5 marks] Write C code for the broadcaster. Use a buffer size of 4096. The process exits when end of input from stdin is hit.

(c) [6 marks] Write C code for the collector. Use `double` for real numbers. Use the "`%g`" format for output. Process exit when end of input from any fanin pipe is hit.

(d) [11 marks] Write C code to fork-exec the n filter children, including redirecting stdin and stdout as necessary.

5. The following are typical system calls involved when using Internet domain (IPv4) stream sockets, listed in random order:

   1. `c = accept(s, NULL, NULL);`
   2. `connect(s, (struct sockaddr *)&youraddr, sizeof(sockaddr_in));`
   3. `s = socket(AF_INET, SOCK_STREAM, 0);`
   4. `bind(s, (struct sockaddr *)&myaddr, sizeof(sockaddr_in));`
   5. `listen(s, 10);`

   Assume that `c` and `s` have been declared, and `myaddr` and `youraddr` have been declared and filled in properly.

   (a) [5 marks] Give the correct order for a server, and indicate which system call should be repeatedly made in a loop if the server anticipates multiple clients. You may write just the numbers or the syscall names.

   (b) [2 marks] Give the correct order for a client (that just needs one connection to one server). You may write just the numbers or the syscall names.

(c) [13 marks] A modern web browser (as a client) makes multiple concurrent connections to servers and downloads from them concurrently. Implement a simple version of this feature.

Two sockets `s1` and `s2` are already created and connected to two servers, and web page requests have already been sent; you just have to read from them now. But use `select` for multiplexing so you can work on any data from either side as soon as it arrives. You may assume `s1` $<$ `s2`. Use a buffer size of 1600 for reading. Since either side may finish much earlier than the other, each socket must be closed as soon as end of input is reached, and should not be given to `select` again.

Two functions are provided to process the data received:

```
void worker1(const char *buf, size_t n);
void worker2(const char *buf, size_t n);
```

Call `worker1` for every chunk of data from `s1`; likewise for `worker2` and `s2`.

(End of questions.)

Technical Reminders

Utility programs. Convention: *file* can be '-' to mean stdin.

- **cat** [*option*]...[*file*]... : Output concatenation of files and/or stdin. With no files, just stdin.
  -b: number non-empty lines
  -n: number all lines

- **head** [*option*]...[*file*]... : Output the first part of files.
  -n *i*: the first *i* lines (default 10)
  -n -*i*; all but the last *i* lines

- **last** [*option*]...[*file*]... : Output the last part of files.
  -n *i*: the last *i* lines (default 10)
  -n +*i*: from line *i* to last

- **sort** [*option*]...[*file*]... : Sort.
  -t *c*: use character *c* as field separator
  -r: reverse sort order
  -n: numeric sort
  -k *m*,*n*: sort by fields from *m* to *n* (as one key)

- **tee** [*option*]...[*file*]... : Copy stdin to stdout and files.
  -a: append to the files instead of overwriting

- **tr** [*option*]... *set1* [*set2*]: Translate, squeeze, delete.
  E.g., 'tr ab 12' converts 'a' to '1', 'b' to '2'.
  -c: complement *set1*
  -d: delete characters that are in *set1*
  -s: replace consecutive occurrences by single occurrence

- **ln** [*option*]... *target linkname*
  ln [*option*]... *target*... *dir*
  Make hard link or symbolic link.
  -s: symbolic link

- **cp** [*option*]... *source dest*
  cp [*option*]... *source*... *dir*
  Copy file(s).
  -r: recursive

- **mkdir** [*option*]... *dir*... : Make directory.
  -p: make intermediate directories as needed

- **basename** *name* : Print *name* with leading directory components removed.

- **dirname** *name*: Print *name* with last non-slash component and trailing slashes removed. If no slash, print "`.`" (meaning the current directory).

Tests in Bourne shell, e.g., `[ -e path ]`:

- `-e path`: path exists
- `-f path`: path exists and regular file
- `-d path`: path exists and directory
- `-r path`: exists and readable (by you)
- `-w path`: exists and writable
- `-x path`: exists and executable
- `-n string`: string is not empty
- `-z string`: string is empty
- `s1 = s2`: string equality
  Also '`!=`', '`<`', '`>`'
- '`n1 -eq n2`': integer equality
  Also '`-ne`', '`-gt`', '`-ge`', '`-lt`', '`-le`'
- `cond1 -a cond2`: and
- `cond1 -o cond2`: or
- `! cond`: not

C preprocessor directives:

- `#include <path>`, `#include "path"`
- `#define` *name* [*replacement*], `#undefine` *name*
- `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`

C library:

- `void *malloc(size_t size);`
- `void *calloc(size_t n, size_t size);`

- `void free(void *ptr);`
- `FILE *fopen(const char *path, const char *mode);`
  modes: r, w, a, r+, w+, a+
- `FILE *fdopen(int fd, const char *mode);`
- `int fclose(FILE *);`
- `int fprintf(FILE *, const char *fmt, ...);`
- `int fscanf(FILE *, const char *fmt, ...);`
  Some formats: `%d` int, `%s` string, `%lf` double
- `size_t fread(void *buf, size_t size, size_t n, FILE *);`
- `size_t fwrite(const void *buf, size_t size, size_t n, FILE *);`

System calls:

- `int open(const char *path, int flags);`
  Some flags: `O_APPEND`, `O_TRUNC`, `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_NONBLOCK`

  If `O_NONBLOCK`, `read` and `write` will not block, instead return -1 and set `errno` to `EAGAIN` or `EWOULDBLOCK`.
- `int fcntl(int fd, F_SETFL, O_NONBLOCK);`
- `int close(int fd);`
- `ssize_t read(int fd, void *buf, size_t n);`
- `ssize_t write(int fd, const void *buf, size_t n);`
- `int execlp(const char *path, const char *arg, ..., (char*)NULL);`
- `pid_t fork(void);`
- `int pipe(int pipefd[2]);`
  `pipefd[0]` is the read end, `pipefd[1]` is the write end
- `int dup2(int oldfd, int newfd);`
- `int select(int n, fd_set *r, fd_set *w, fd_set *e, struct timeval *timeout);`
  n is 1 + your highest file descriptor
- `void FD_ZERO(fd_set *s);`
  `void FD_SET(int fd, fd_set *s);`
  `void FD_CLR(int fd, fd_set *s);`
  `int FD_ISSET(int fd, fd_set *s);`

Blank page for sketch work. You may tear out for convenience; no need to hand in.

Blank page for sketch work. You may tear out for convenience; no need to hand in.