Full Name:__________________________________

Concurrent & Distributed Systems (CS 475)
Spring 2010
Final Exam

Instructions

- The exam has a maximum score of 100 points. You have TWO HOURS to finish this exam.
- Make sure that your exam is not missing any sheets, and then write your full name on the front.
- Write your answers in the space provided below the problem.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is OPEN BOOK. You may use any books or notes you like. Good luck!

Problem 1 (15 pts)
This problem tests your understanding of concurrent programming and exceptional control flow. For each of the following programs, list **all** possible outputs in a comma-separated list. Assume that system calls execute without error.

Program 1

```
int i = 0;

int main() {
      if(fork() == 0)
            i++;
      else
            i+=2;
      printf("%d",i);
}
```

All possible outputs:

Program 2

```
int i = 0;

int main() {
      if(fork() == 0)
            i++;
      else {
            i+=3;
            wait(NULL);
      }
printf("%d",i);
}
```

All possible outputs:

Program 3
```
int i = 0;

void *doit(void *vargp)
{
      pthread_detach(pthread_self());
      i++;
}

int main()
{
      pthread_t tid;
      pthread_create(&tid, NULL, doit, NULL);
      i++;
      printf("%d",i);
}
```

All possible outputs:

Program 4
```
int i = 0;

void *doit(void *vargp) {
      i = i + 5;
}

int main()  {
      pthread_t tid;
      ptr = &i;
      pthread_create(&tid, NULL, doit, NULL);
      i = i + 3;
      pthread_join(tid, NULL);
      printf("%d",i);
}
```

All possible outputs:

Program 5
```
void *doit(void *vargp) {
      int i = 3;
      int *ptr = (int*)vargp;
      (*ptr)++;
}

int main() {
      int i = 0;
      pthread_t tid;
      pthread_create(&tid, NULL, doit, (void*)&i);
      pthread_join(tid,NULL);
      i = i + 4;
      printf("%d",i);
}
```

All possible outputs:

Problem 2 (20 pts)
We explore the so-called barbershop problem. A barbershop consists of a n waiting chairs and the barber chair. If there are no customers, the barber waits. If a customer enters, and all the waiting chairs are occupied, then the customer leaves the shop. If the barber is busy, but waiting chairs are available, then the customer sits in one of the free chairs.

Here is the skeleton of the code, without synchronization.

```
extern int N; /* initialized elsewhere to value > 0 */
int customers = 0;

void* customer() {



  if (customers > N) {



     return NULL;
  }



  customers += 1;



  getHairCut();



  customers -= 1;



  return NULL;
}

void* barber() {
  while(1) {



     cutHair();



   }
}
```

For the solution, we use three binary semaphores:

- `mutex` to control access to the global variable customers.
- `customer` to signal a customer is in the shop.
- `barber` to signal the barber is busy.

1. Indicate the initial values for the three semaphores.

    - `mutex`

    - `customer`

    - `barber`

2. Complete the code above filling in as many copies of the following commands as you need, but no other code.

```
P(&mutex);
V(&mutex);
P(&customer);
V(&customer);
P(&barber);
V(&barber);
```

Problem 3 (30 pts)

Part of the code for the web server TINY (from the book Computer Systems: A Programmer's Approach) is shown below. Specifically, the code for the functions `main()`, `doit()` and `read_requesthdrs()` is provided.

The TINY web server only implements the functionality for the GET method of HTTP. Your job is to extend the code in the function doit() so that TINY can handle the POST method in addition to GET.

For this problem, you are only required to **write the code for reading the POST request message**, i.e. for reading the HTTP message corresponding to the POST request (including the request headers as well as the body of the message). **You do not need to write the code for processing the POST message after it has been read.**

```c
/*
 * tiny.c - A simple, iterative HTTP/1.0 Web server that uses the
 *     GET method to serve static and dynamic content.
 */
#include "csapp.h"

void doit(int fd);
void read_requesthdrs(rio_t *rp);
int parse_uri(char *uri, char *filename, char *cgiargs);
void serve_static(int fd, char *filename, int filesize);
void get_filetype(char *filename, char *filetype);
void serve_dynamic(int fd, char *filename, char *cgiargs);
void clienterror(int fd, char *cause, char *errnum,
             char *shortmsg, char *longmsg);

int main(int argc, char **argv)
{
    int listenfd, connfd, port, clientlen;
    struct sockaddr_in clientaddr;

    /* Check command line args */
    if (argc != 2) {
      fprintf(stderr, "usage: %s <port>\n", argv[0]);
      exit(1);
    }
    port = atoi(argv[1]);

    listenfd = Open_listenfd(port);
    while (1) {
      clientlen = sizeof(clientaddr);
      connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
      doit(connfd);
      Close(connfd);
   }
}
```

```
void doit(int fd)
{
    int is_static;
    struct stat sbuf;
    char buf[MAXLINE], method[MAXLINE], uri[MAXLINE], version[MAXLINE];
    char filename[MAXLINE], cgiargs[MAXLINE];
    rio_t rio;

    /* Read request line and headers */
    Rio_readinitb(&rio, fd);
    Rio_readlineb(&rio, buf, MAXLINE);
    sscanf(buf, "%s %s %s", method, uri, version);
    if (strcasecmp(method, "GET")) {
        clienterror(fd, method, "501", "Not Implemented",
                "Tiny does not implement this method");
        return;
    }
    read_requesthdrs(&rio);

    /* Parse URI from GET request */
    is_static = parse_uri(uri, filename, cgiargs);

    /* if request for static content call serve_static()
    /* otherwise call serve_dynamic()
    /*
    /*  THE CODE IS NOT SHOWN HERE SINCE IT IS NOT NEEDED FOR
    /*  ANSWERING THIS QUESTION  */

}
```

```
/*
 * read_requesthdrs - read and parse HTTP request headers
 */

void read_requesthdrs(rio_t *rp)
{
    char buf[MAXLINE];

    Rio_readlineb(rp, buf, MAXLINE);
    while(strcmp(buf, "\r\n")) {
      Rio_readlineb(rp, buf, MAXLINE);
      printf("%s", buf);
    }
    return;
}
```

In the space below on this page, explain clearly the changes (if any) that need to be made to the `functions main(), doit()` and `read_requesthdrs()` for TINY to be able to read the HTTP message corresponding to a POST request. **Clearly explain the logic behind your approach.** On the next two pages, write down the new code for any functions that you need to modify or extend. **Note that you are required to use the RIO functions for reading and writing to socket filedescriptors.**

Problem 4 (5 pts)
Consider again the web server TINY from the previous question. TINY is an iterative web server. You have been asked to re-implement TINY as a concurrent server that creates a new child process for each new connection request. In the space below, show how you would modify the main() and doit() routines of TINY for this purpose.

Problem 5 (30 pts)

Consider a banking service that provides two interfaces -- `Account` and `Branch`, as shown below. Each account is represented by a remote object whose interface `Account` provides operations for making deposits and withdrawals and for enquiring about and setting the balance. Each branch of the bank is represented by a remote object whose interface `Branch` provides operations for creating a new account, for looking up an account by name and for enquiring about the total funds at the branch.

```
// Operations of the Account Interface

deposit(amount)
        deposit amount in the account
withdraw(amount)
        withdraw amount from the account
getBalance() ----> amount
        return the balance of the account
setBalance(amount)
        set the balance of the account to amount
```

```
 // Operations of the Branch interface

create(name) ----> account
        create a new account with a given name
lookup(name) ----> account
        return a reference to the account with a given name
branchTotal() ----> amount
        return the total of all the balances at the branch
```

PART 1:

Suppose you were asked to implement the Account and Branch classes using Java RMI. On the next two pages, write down Java interfaces for the Account and Branch classes respectively. Assume that the data types of the parameters `name` and `amount` are string and double respectively.

PART 2.
Java RMI provides at most once invocation semantics. However, suppose you were informed that the RMI system being used provided **at least once** invocation semantics. Would the interfaces for Account and Branch need to be changed to take into account the difference in invocation semantics? If so, what changes would need to be made to the interfaces for Account and Branch (shown on page 11), **and explain why**. If not, **explain why** the existing interfaces are adequate even in a RMI system that provides at least once semantics.