Question 1. [10 MARKS]

Circle the correct answer for the following questions:

TRUE [✓ FALSE] A process running in the background is an orphaned process.

TRUE [✓ FALSE] Signals are used to send several bytes of data from one process to another process.

[✓ TRUE] FALSE A process is in the blocked state if it is waiting for a child to terminate (i.e., it has called `wait(&status)`).

TRUE [✓ FALSE] An inode stores the contents of a file.

TRUE [✓ FALSE] An inode contains a process id.

[✓ TRUE] FALSE A signal can be generated by a software or hardware event.

TRUE [✓ FALSE] The kill command is only used to terminate processes.

TRUE [✓ FALSE] The following two commands will display the same list of file names:
`find . -name "a209*" -print`,
`ls ./a209*`

[✓ TRUE] FALSE A new process is created in Unix only when a running process calls `fork()`

.

Question 2. [5 MARKS]

What is the output of the following set of Bourne shell commands, given that the current working directory contains the following files? (’ is a forward quote, and ‘ is a backward quote).

```
a.c    b.h    d.ch   f.c
```

cmd=”ls”

`echo $cmd *`	`ls a.c b.h d.ch f.c`
`echo *.[ch]`	a.c b.h f.c
`echo "$cmd *.c"`	`ls *.c`
`echo ’$cmd *.c’`	`$cmd *.c`
`echo ‘$cmd *.c‘`	`a.c f.c`

Question 3. [9 MARKS]

In the program below, assume that all processes run until they terminate normally, and the shell that executes the program does not terminate.

```
int main() {
    int status, p1, p2;
    p1 = fork(); /*child 1*/

    if(p1 == 0) {
        printf("A\n");
        p2 = fork();/*child 2*/
        if(p2 == 0) {
            printf("C\n");
        } else {
            printf("D\n");
        }
        wait(&status);

    } else {
        printf("B\n");
    }

    printf("All done\n"); exit(0);
}
```

Part (a) [3 MARKS]

Write the output of this program in a valid order.

B
A
D
All done
C
All done
All done

Part (b) [3 MARKS]

For each process in the above program, state whether it is possible for the process to become a zombie. If it is possible for the process to become a zombie, explain the circumstances under which it could become a zombie, otherwise explain why the process could not become a zombie.

Child 1 would become a zombie if it terminates or is killed before the parent terminates. Child 2 would become a zombie if it terminates before Child 1 calls wait.

Part (c) [3 MARKS]

For each process in the above program, state whether it is possible for the process to become an orphan. If the process can become an orphan explain the circumstances under which it would be orphaned, otherwise explain why the process could not become an orphan.

Child 1 could become an orphan if the parent terminates before it does. Child 2 could not become an orphan because it's parent calls wait, so the child is guaranteed to terminate before the child. The parent could not become an orphan because the shell that executes it does not terminate.

Question 4. [6 MARKS]

When the following Perl patterns are applied to the string `$s1`, write the value of `$1` if the pattern matches the string. If the pattern does not match the string, write “undef”

```
$s1 = "Don’t Cry Joe 3:03 1955";
```

`/(\w+)/`	`$1 =`	Don
`/(\w+)$/`	`$1 =`	1955
`/(\d+)/`	`$1 =`	3
`/(\d:\d\d)$/`	`$1 =`	undef
`/([\d:]+)/`	`$1 =`	3:03
`/([\w\s]+)\s+\d+:\d+\s*\d?/`	`$1 =`	t Cry Joe

Question 5. [12 MARKS]

Part (a) [3 MARKS]

List three types of software tools, other than an editor and a compiler, that a professional programmer should be able to use well, and give a one-sentence description of the purpose of each.

- version control system
- build system (e.g. make)
- debugger
- profiler
- regression testing
- issue tracking

Part (b) [4 MARKS]

A programmer chooses a language based on the type of problem to be solved. Give one reason why a programmer might choose a compiled language over an interpreted language, and one reason why a programmer might choose an interpreted language over a compiled language for a particular problem.

A compiled language can lead to faster code since the compiler can perform global optimizations.

An interpreted language is good for short programs or prototypes that do not need to be high-performance since there are usually many machine level instructions for each line of interpreted code. Also, the compile debug cycle is shortened.

Part (c) [5 MARKS]

Semaphores may be used keep track of the references to some resource (e.g., a file, or a shared data structure). The creator of the resource also creates and initializes the associated semaphore. The creator uses the semaphore to determine when it is safe to delete the resource and the semaphore. It is safe to delete a resource when there are no more references to it. Other processes use the sempahore to indicate when they are actively using the resource (e.g., they have a file pointer to an open file, or a pointer to a shared data structure).

Explain, in English, the semaphore operations required 1) by the creator of the resource and 2) by other processes using the resource. Be clear about when processes might block, and the value of the semaphore variable at each stage.

Creator: Creates and initializes the semaphore to 0. When it deletes the semaphore, it blocks until the value of the semaphore is 0, and then deletes the semaphore.

Other processes: Gets a semaphore. When a process is actively using the resource it increments the semaphore variable, and decrements the semaphore variable when it is finished using the resource. Other processes never block on a semaphore operation.

Question 6. [10 MARKS]

Write a C program that uses `fork` and `pipe` to set up the following pipeline and execute it.

```
grep a209 /u/csc209h/submitted | wc
```

```
int
main()
{
    int fd[2];
    char buf[10];

    pipe(fd);

    if((fork()) == 0) {

        dup2(fd[1], fileno(stdout));
        close(fd[0]); close(fd[1]);
        execlp("/usr/ucb/grep", "grep", "a209",
               "/u/csc209h/submitted", 0);
        perror("exec");
    } else {
        dup2(fd[0], fileno(stdin));

        close(fd[0]); close(fd[1]);

        execlp("/usr/bin/wc","wc", (char *)0);
        perror("exec");
    }
}
```

Question 7. [12 MARKS]

A given remote data server receives a connection request from a client, and reads a file name from the client. Then it writes the contents of the file to the client, one line at a time (a line will be no longer than `LINE_SIZE`). When it finishes sending the file, the server closes the connection.

Complete the following socket **client** program that sends requests to the remote data server for the files given in `names`, creates the file, and writes the lines sent from the server to the file.

You must ensure that the client closes the socket, even when an error occurs. Recall that `htons()` must be called to convert a port into network byte order, and that the socket type will be `AF_INET`.

```
#define SERVER_PORT  30000
#define LINE_SIZE 16

char *names[5] = {"afile", "bbfile", "cfile", "df", "efile"};
char *hostname = "penguin";

int main(void) {
    struct hostent *hp = gethostbyname(hostname);
```

```
    int i;    FILE *fp;    int soc;    char buf[BLOCK_SIZE];
    struct sockaddr_in peer;

    peer.sin_family = AF_INET;
    peer.sin_port = htons(SERVER_PORT);
    peer.sin_addr = *((struct in_addr *)hp->h_addr);

    for(i = 0; i < 5; i++) {
        soc = socket(AF_INET, SOCK_STREAM, 0);
        if (connect(soc, (struct sockaddr *)&peer, sizeof(peer)) == -1) {
            perror("client:connect"); close(soc);
            exit(1);
        }

        write(soc, names[i], strlen(names[i]));

        if((fp = fopen(names[i], "w")) == NULL) {
            perror("fopen");
            close(soc);
            break;
        }
        while(read(soc, buf, sizeof(buf))) {
            printf("%s\n", buf);
            fprintf(fp, "%s", buf);
        }
        close(soc);
        fclose(fp);
    }
}
```

Question 8. [12 MARKS]

Part (a) [8 MARKS]

Write a C program that creates 4 processes. Each of the four processes calls the function `do_child()` defined below, and returns the result of `do_child()` as its termination code. The parent process will print to standard output the value of each of the child's termination code. If a child process receives a SIGTERM signal the child should call a signal handler that terminates the child process, passing the current value of `(result/count)` as the child's termination code. The parent process should not modify the behaviour of any signals. Several global variables have been declared for you. You may need to write one or more functions in addition to main.

Part (b) [4 MARKS]

Modify `do_child()` so that it blocks SIGTERM while it is executing the two lines marked with an `X`. Write the additional code beside the function `do_child()` and indicate with arrows where each piece of the added code should go.

```
int child_result;
double result;
int count;

int do_child( void ) {         | sigset_t tset;
    result = 0.0;           <--| sigemptyset(&tset);
    count = 0;                 | sigaddset(&tset, SIGTERM);

    while(count < 10) {
                            <-- sigprocmask(SIG_BLOCK, &tset, NULL);
        result += runexp();   /*X*/
        count++               /*X*/
                            <-- sigprocmask(SIG_UNBLOCK, &tset, NULL);
    }
    return result / count;
}
```

(*Complete part (a) below*)

An extra page if you need it.

```
void quit(int code) {
    exit(child_result);
}

int main(void) {
    int res, i, pid, status; int total;
    struct sigaction sa, old;

    for(i = 0; i < 3; i++) {

        if((pid = fork())== 0) {
            sa.sa_handler = quit;
            sigemptyset(&sa.sa_mask);
            sa.sa_flags = 0;

            res = sigaction(SIGTERM, &sa, &old);
            child_result = do_child();
            exit(child_result);
        }
    }

    for(i = 0; i < 3; i++) {
        if((pid = wait(&status)) > 0) {
           total = WEXITSTATUS(status);
           printf("Result from %d = %d\n", pid, total);
        }
    }
}
```

Question 9. [14 marks]

Assume you are given the source code and a Makefile which compiles an executable called `updateFile`. The specifications of `updateFile` are given below. Your job is to write two programs that can be used to test `updateFile`: a Perl program that will check if the output of `updateFile` follows the specifications, and a Bourne shell program that runs the tests. Do **not** write `updateFile`. You may assume that the source code and Makefile to produce the executable `updateFile` are in the current working directory.

The program `updateFile` creates or appends to a file the output from the parent and a child process. It is run as "`updateFile -p <pl> -c <cl> -f <fname>`" where `<pl>` is the number of lines the parent process writes to the file `<fname>` and `<cl>` is the number of lines the child process writes to `<fname>`. A line written by the parent begins with "Parent", and a line written by the child begins with "Child". `updateFile` uses a semaphore to synchronize the writes to the file.

Part (a) [5 marks]

Write a Perl program that takes two arguments and reads from **standard input**. The first argument is the number of lines of input expected to begin with the word "Parent" and the second argument is the number of lines expected to begin with the word "Child". It will print "Correct" to standard output if it found the correct number of lines beginning with "Parent" and "Child", and "Wrong" otherwise.

```
#!/local/bin/perl

my $np = $ARGV[0];
my $nc = $ARGV[1];

my $sumParent = 0;
my $sumChild = 0;
while(<STDIN>) {

  if(/^Parent/) {
    $sumParent++;
  }
  if(/^Child/) {
    $sumChild++;
  }
}

if($sumParent == $np && $sumChild == $nc) {
  print "Correct";
} else {
  print "Wrong";
}
```

Part (b) [9 marks]

Write a Bourne shell program to test `updateFile`. The shell program uses the Perl program in part a) which is stored in a file called `matchOut` to check if the output produced by `updateFile` conforms to the specifications.

The shell program produces output that answers the following questions. Assume all necessary files are in the current working directory:

- Does the makefile correctly produce the executable?
- If the file “afile” exists, does “updateFile -p 5 -c 10 -f afile” produce the expected output? Assume “afile” contains 3 lines beginning with “Parent” and 4 lines beginning with “Child”.
- If the file “bfile” does not exist does “updateFile -p 1 -c 1 -f bfile” produce the expected output?
- Does updateFile remove the semaphore when it terminates? (Assume that the user is not running any other programs requiring semaphores.) If the semaphore is not removed, the shell program should remove it.

Some commands you will find useful are “`whoami`” which prints the name of the user, “`ipcrm sem <key>`” which removes a semaphore, and “`ipcs -s`” which produces output of the form:

```
key        semid      owner      perms      nsems      status
0x0006f385 5177350    a209xxxx   666         1
```

```
make

if [ ! -x updateFile ]
then
    echo "cannot find updateFile"
    exit
fi

touch afile
echo "Parent foo
Child blah
Child blah
Child blah
Parent Child
Child Parent
Parent Parent" > afile

rm bfile

updateFile -p 5 -c 10 -f afile
matchOut 8 14 < afile

updateFile -p 1 -c 1 -f bfile
matchOut 1 1 < bfile
```

```
# (continued)
name=‘whoami‘
sem=‘ipcs -s | grep $name‘

if [ -z "$sem" ]
then
    echo "Semaphore removed"
else
    echo "Semaphore not removed"
fi
```