CS39002 Operating Systems Laboratory
Lab Test 1
03-Feb-2026, 6:15pm – 7:30pm
Maximum Marks: 50

Roll No: _____

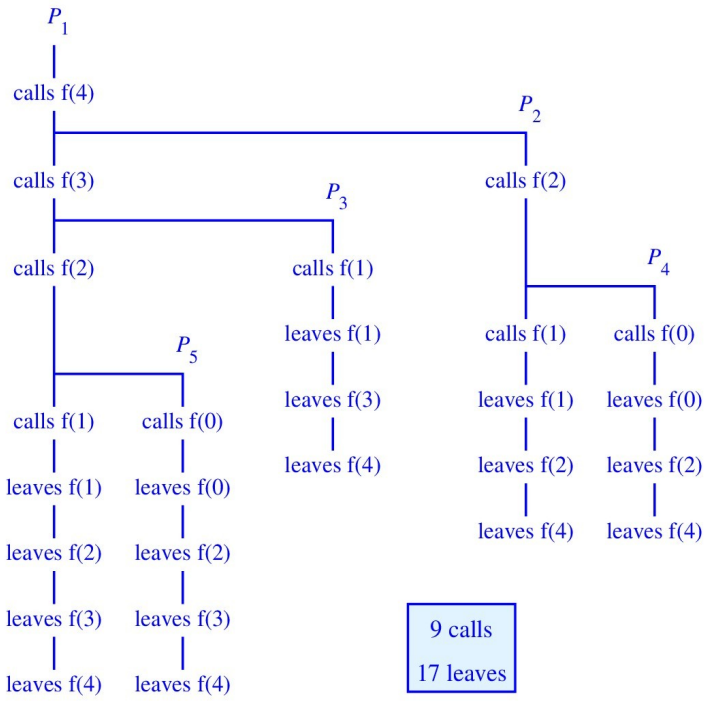
Name: _____

1. Consider the following recursive function in a C program.

```
void f ( int n )
{
   printf("Process with PID = %d calls f(%d)\n", getpid(), n); fflush(stdout);
   if (n >= 2) {
      if (fork()) f(n-1); else f(n-2);
   }
   printf("Process with PID = %d leaves f(%d)\n", getpid(), n); fflush(stdout);
}
```

The user supplies a non-negative integer value of `n` as the command-line argument. The `main()` function only calls `f(n)`, and exits. You compile the program (to `a.out`), and run the following two shell commands. Here, `grep` searches for the pattern "`calls`" or "`leaves`" in the lines printed by the runs of `a.out`, and `wc -l` prints the number of lines that match the given pattern. Derive with proper justifications what output you will get for these two shell commands. Name the processes as $P_1, P_2, P_3, \ldots$ List the sequence of calls and leaves (like "`calls f(4)`" or "`leaves f(0)`") under each process (in actual order). Make a bifurcation when `fork()` creates a child process. **[8]**

```
$ ./a.out 4 | grep "calls" | wc -l
$ ./a.out 4 | grep "leaves" | wc -l
```



This is not asked in the question, but here are some interesting results. Consider the following sequences.

$P(n)$ = Total number of processes involved by the call $f(n)$ (including the outermost process)
$C(n)$ = Total number of all calls of $f(\,)$, initiated by the outermost call of $f(n)$
$L(n)$ = Total number of all leaves of $f(\,)$, initiated by the outermost call of $f(n)$

We have the following recurrences (establish their correctness).

$P(0) = P(1) = 1$, and $P(n) = P(n-1) + P(n-2)$ for $n \geq 2$.
$C(0) = C(1) = 1$, and $C(n) = 1 + C(n-1) + C(n-2)$ for $n \geq 2$.
$L(0) = L(1) = 1$, and $L(n) = L(n-1) + L(n-2) + P(n)$ for $n \geq 2$.

2. The game of tic-tac-toe is played on a 3 × 3 board. The game involves three processes: *umpire*, *player X*, and *player O*. A sample run of the game is shown below (right). The implementation details are now elaborated.

```
$ ./umpire
        Player X starts
        Player O starts
   |   |
---+---+---
   |   |
---+---+---
   |   |

[X] 1 0
   |   |
---+---+---
 X |   |
---+---+---
   |   |

[O] 1 1
   |   |
---+---+---
 X | O |
---+---+---
   |   |

[X] 2 2
   |   |
---+---+---
 X | O |
---+---+---
   |   | X

[O] 0 2
   |   | O
---+---+---
 X | O |
---+---+---
   |   | X

[X] 2 0
   |   | O
---+---+---
 X | O |
---+---+---
 X |   | X

[O] 0 0
 O |   | O
---+---+---
 X | O |
---+---+---
 X |   | X

[X] 0 2
Invalid move…
[X] 0 3
Invalid move...
[X] 2 1
 O |   | O
---+---+---
 X | O |
---+---+---
 X | X | X

Player X wins
        Player X leaving
        Player O leaving
Players exited
Bye...
$
```

The cells of the board are indexed in the row-major order in a two-dimensional 3 × 3 array B of `char` variables. The top left cell has index (0, 0), and the bottom right cell has index (2, 2). At the beginning, each cell is initialized to the space character.

You write two C program files *umpire.c* and *player.c*. The first file implements the process *umpire*, whereas the second file implements the behavior of each *player* (*X* and *O*). The *umpire* coordinates the moves of the game. The *player*s interactively (that is, from the terminal) and alternately enter the cell numbers (row indices and column indices as per the above convention) which they want to mark. The user inputs are shaded in the sample run to the right.

The *umpire* (the executable from *umpire.c*) starts by creating **a single pipe** P for **all** inter-process communications. It then forks two child processes (one for each *player*), each of which `exec`s the executable file *player* generated by compiling *player.c*. The *umpire* will only read from the pipe P, whereas each *player* will only write to P. The `exec` system call should send two command-line parameters: the player (*X* or *O*), and the descriptor of the write end of P. Each *player* starts by printing a message to the terminal (indented as shown at the beginning of the sample run).

Each move of the game involves the following interaction among the processes. Both *player*s wait for a signal from the *umpire*. The *umpire* keeps track of which *player* should make the next move; call it N (where N is either *X* or *O*). The *umpire* prints the prompt [N] (to the terminal), sends SIGUSR1 to N, and waits to read from the pipe P. The *player* N jumps to its SIGUSR1 handler function `nextmove()`. It reads from the user (terminal) the two indices of the cell which it wants to mark, and writes these indices to the pipe P. The *umpire* reads (from P) and validates the indices. A cell (*row*, *col*) is invalid if *row* and/or *col* is/are not in the range [0, 2], or the cell is already marked by some player. So long as the indices sent by N are invalid, the *umpire* keeps on reading from N. When a valid cell is supplied by N, the *umpire* marks that cell by N.

After this, the *umpire* calls a function `checkboard`(B). If B is a winning situation for *player X* or *O*, the function returns the character `X` or `O` accordingly. Otherwise it returns the space character. If the *umpire* receives a non-space character from `checkboard()`, it prints (to the terminal) "`Player N wins`". Otherwise, if the board is not full, the game continues, or if the board is full, the *umpire* prints (to the terminal) "`The game ends in a draw`". If the game ends in a win or a draw, the *umpire* calls the function `endgame()`. This function is also called if the user hits control-C (see below).

Both *umpire* and *player* should catch the signal SIGINT (control-C). For the *umpire*, the SIGINT handler is the function `endgame()` mentioned above. For each *player*, the SIGINT handler is a function `leave()`. In `endgame()`, the *umpire* sends SIGINT to both *player*s, waits for their termination (use `waitpid`), prints a message (see the end of the sample to the right), and itself exits. In `leave()`, each *player* prints a message to the terminal (indented in the sample), and exits.

All reads and all writes should use the high-level I/O calls `scanf`/`printf`. Use appropriate `dup()` calls so that the `stdin` and `stdout` buffers are redirected as required. Flush the `stdout` buffer after every write. Do **not** use `stderr`. Keep a copy of the terminal I/O handlers (and reinstate them) whenever needed.

On the next two pages, fill out the details to implement the behavior of the processes as described above. Use C constructs only. Do **not** use `sleep()` or `usleep()` anywhere except only at the beginning of *umpire*.

(a) The code for *player.c*

First, write the `main()` function. Notice that each *player* is supplied two command-line arguments: the identity of the player (X or O), and the descriptor of the write end of the pipe P. Two global variables `player` (type `char`) and `pipefd` (type `int`) are set from these. Then the SIGUSR1 and the SIGINT handlers are registered. After other bookkeeping (like the necessary `dup()` calls), the *player* should go to an infinite wait-for-signal loop.

```
int main ( ____________________________int argc, char *argv[]____________________________ )
{
```

[10]

```
    player = argv[1][0];
    pipefd = atoi(argv[2]);

    signal(SIGUSR1, nextmove);
    signal(SIGINT, leave);

    printf("\tPlayer %c starts\n", player);

    /* player needs to reuse the terminal for printing the final message, so keep a copy of fd 1 */
    dupout = dup(1);  /* dupout is a global variable of type int */
    close(1); dup(pipefd);

    while (1) pause();

}
```

Now, write the signal handler `nextmove`() for SIGUSR1, and the signal handler `leave`() for SIGINT. **[8]**

```
void nextmove ( int sig )
{
   int row, col;

   scanf("%d%d", &row, &col);
   printf("%d %d\n", row, col);
   fflush(stdout);
}


void leave ( int sig )
{
   close(1);
   dup(dupout);
   printf("\tPlayer %c leaving\n", player);
   exit(0);
}
```

(b) The code for *umpire.c*

First, write the signal handler `endgame`() for SIGINT. Note that the umpire does not catch SIGUSR1. Its blocking waits are effected by reading from the pipe *P*, and (at the end) by `waitpid`(). **[4]**

```
void endgame ( int sig )
{
   kill(xpid, SIGINT); waitpid(xpid, NULL, 0);
   kill(opid, SIGINT); waitpid(opid, NULL, 0);
   printf("Players exited\nBye...\n");
   exit(0);
}
```

The tic-tac-toe board is maintained as 3 × 3 array `B` of `char`. You do **not** need to write the following two functions on `B`: a function `printboard`(`B`) to print the 3 × 3 array as in the sample run, and the function `checkboard`(`B`) as explained on the last page. Only call these functions at appropriate places.

In the `main()` function, the pipe *P* is created, two *player* processes *X* and *O* are forked (their PID's are stored as `xpid` and `opid`, and they `exec` *player* with appropriate command-line arguments), the handler `endgame()` is registered, and appropriate `dup()` calls are made. After a second's delay (to allow the child processes to be active in the system), a call to the function `rungame()` is made. This function does not return to `main()`. The *umpire* exits from the function `endgame()`.

```
int main ( )      /* No command-line arguments for umpire */                    [10]
{
```

```
    int pfd[2];
    char pipefd[16];
    char B[3][3] = { { ' ', ' ', ' ' }, { ' ', ' ', ' ' }, { ' ', ' ', ' '} };

    pipe(pfd);
    sprintf(pipefd, "%d", pfd[1]);

    xpid = fork();
    if (!xpid) execlp("player", "./player", "X", pipefd, NULL);

    opid = fork();
    if (!opid) execlp("player", "./player", "O", pipefd, NULL);

    signal(SIGINT, endgame);

    /* umpire will never read from terminal, so permanently close fd 0 */
    close(0); dup(pfd[0]);

    sleep(1);
    rungame(B, xpid, opid);

    exit(0);
```

```
}
```

Finally, write the function `rungame()` that enters a loop. Each iteration of the loop carries out one move of the game, as explained earlier in detail. In short, the body of the loop involves sending SIGUSR1 to the next player *N*, reading from *N* (and validating the data), marking the specified cell in `B`, printing the updated board, checking for possible win/draw status of the game, and (if needed) calling `endgame()`.

```
void rungame ( ____________________ char B[][3] , int xpid, int opid ____________________ )   [10]
{
```

```
    int row, col, nmove = 0;
    char player, winner;
    int plrpid;

    printboard(B);
    while (1) {
       player = (nmove % 2 == 0) ? 'X' : 'O';
       plrpid = (nmove % 2 == 0) ? xpid : opid;
       printf("[%c] ", player); fflush(stdout);
       kill(plrpid, SIGUSR1);
       scanf("%d%d", &row, &col);
       if ((row < 0) || (row > 2) || (col < 0) || (col > 2) || (B[row][col] != ' ')) {
          printf("Invalid move...\n");
          continue;
       }
       ++nmove;
       B[row][col] = player;
       printboard(B);
       winner = checkboard(B);
       if (winner == player) { printf("Player %c wins\n", player); break; }
       if (nmove == 9) { printf("Game ends in a draw\n"); break; }
    }
    endgame(SIGINT);
```

```
}
```