CS39002 Operating Systems Laboratory
Lab Test 1
07-Feb-2025, 7:30pm – 8:30pm
Maximum Marks: 40

Roll No:	______________________
Name:	______________________

1. Consider the following function. Assume that the argument n supplied to the function is non-negative.

```
void f ( int n )
{
   int i;

   for (i=0; i<n; ++i) {
      if (fork()) { printf("A\n"); fflush(stdout); }
   }
   for (i=0; i<n; ++i) wait(NULL);
   printf("B\n"); fflush(stdout);
   exit(0);
}
```

(a) Derive, as a function of n, how many `A`'s are printed by the call `f(n)`. Give proper justification. **[4]**

In the first iteration, the parent process P prints one A, and creates a new child process C. Both P and C run the loop for $n - 1$ more iterations. The number $A(n)$ of A's printed by $f(n)$ therefore satisfies:

$$A(n) = 2\,A(n-1) + 1 \quad \text{for} \quad n \geqslant 1.$$

Moreover,

$$A(0) = 0.$$

This is the Tower-of-Hanoi recurrence with the solution

$$A(n) = 2^n - 1 \quad \text{for all} \quad n \geqslant 0.$$

(b) Derive, as a function of n, how many `B`'s are printed by the call `f(n)`. Give proper justification. **[4]**

The number of B's printed is equal to the number of processes (including the process that makes the call $f(n)$). The parent runs the loop for $i = 0, 1, 2, \ldots, n-1$, with the i-th iteration creating a new child which runs the same loop for $n - i - 1$ iterations. The number $B(n)$ of B's printed by $f(n)$ therefore satisfies:

$$B(n) = B(n-1) + B(n-2) + \ldots + B(0) + 1 \quad \text{for } n \geqslant 1.$$

Moreover,

$$B(0) = 1.$$

By induction, it follows that

$$B(n) = 2^n \quad \text{for all} \quad n \geqslant 0.$$

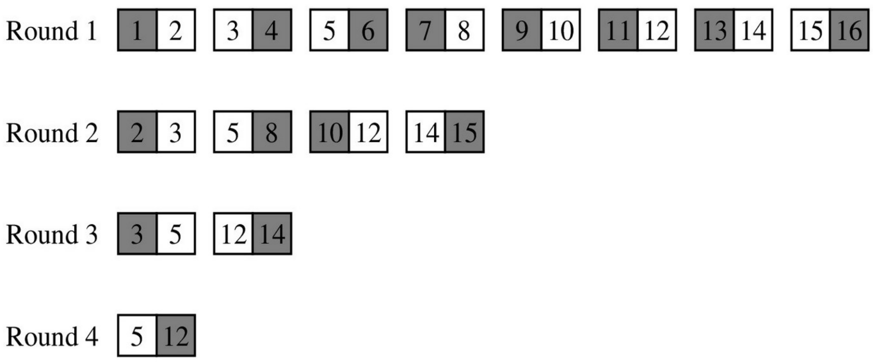
You may also argue that the number of processes for the call $f(n)$ is twice the number of processes for the call $f(n-1)$.

(c) Prove/Disprove with proper justification: For all values of `n` $\geqslant 0$, the call `f(n)` necessarily prints all the `A`'s before all the `B`'s (irrespective of how the processes are scheduled). **[4]**

False. Consider the call of $f(1)$. The parent process P forks a child process C in the only iteration of the loop. Suppose that immediately after the call of *fork*(), P is preempted. Subsequently, C is scheduled. C does not make any iteration of the loop, prints B, and exits. Then, P is rescheduled. It prints A, goes out of the loop, prints B, and exits. So the printing sequence in this case is:

B
A
B

2. Consider a tournament played in r rounds by $n = 2^r$ players. The players are numbered 1, 2, 3, . . . , n. In the first round, Player 1 plays with Player 2, Player 3 plays with Player 4, and so on. The winners enter the second round. Winner 1 plays with Winner 2, Winner 3 plays with Winner 4, and so on. Round 3 is played by the winners of Round 2, and so on. In the last round, only two remaining players play. A sample tournament with $r = 4$ (and $n = 16$) is shown in the picture below. The losers are shaded. After $r = 4$ rounds, Player 5 becomes the winner, whereas Player 12 ends up as the runner-up.



The outcomes of the matches are decided by a program called *umpire*. Each player is simulated by the program *player*. The program *umpire* first reads r from the user (or the command line), and calculates $n = 2^r$. It then forks n child processes. Each child process execs *player* with two command-line arguments: the number p of the player, and the number r of rounds. After this, *umpire* simulates all the matches randomly, round by round (in the sequence Round 1, Round 2, . . . , Round r). If w is the winner, and l the loser of a match, *umpire* sends SIGUSR1 to w and SIGUSR2 to l. The loser exits immediately (except in Round r). The winner continues to run. After the last match (the only match in the r-th round), the winner W prints "Player W: I am the winner", and the loser L prints "Player L: I am the runner-up", and both *player*s W and L exit.

You do not have to write the code for *umpire*. Write the code for each *player* below. Use only C constructs. Avoid OS-specific system calls (like Linux-specific *sigaction*()).

(a) Write the *main*() function of *player*. Note that *player* should avoid busy waits by using *pause*(). Each *player* maintains three global variables: p (the player number), r (the number of rounds), and *round* (the current round). It should catch SIGUSR1 and SIGUSR2. For both these signals, the same signal-handler *match*() is to be used. You do not have to synchronize the parent process (*umpire*). **[7]**

```
int main ( int argc, char *argv[] )
{
    /* Read p and r from command-line arguments */

    p = atoi(argv[1]);
    r = atoi(argv[2]);

    /* Initialize round */

    round = 0;

    /* Register the signal handler */

    signal(SIGUSR1, match);
    signal(SIGUSR2, match);

    /* Enter into a non-busy wait */

    While (1) pause();

    exit(0);
}
```

(b) Now, write the signal-handler function *match*(). This should be the only function (other than *main*()) in the *player* program. This would handle both SIGUSR1 and SIGUSR2. The behavior of *player* upon the reception of the signals is explained below the picture on the last page. Use no variables other than the global variables *p*, *r*, and *round* as described earlier. **[7]**

```
   void        match (            int sig             )
{

   /* signal received for another round */
   ++round;

   /* response to the signal depends on the type of the signal */
   if (sig == SIGUSR1) {
      /* player continues to play unless it is the last round */
      if ( round == r ) {
         printf("Player %d: I am the winner\n", p);
         exit(0);
      }
   } else if (sig == SIGUSR2) {
      /* player prints runner-up notification in only the last round, and exits anyway */
      if ( round == r )
         printf("Player %d: I am the runner-up\n", p);
      exit(0);
   }

}
```

3. Suppose that a parent process P and n child processes $C_0, C_1, \ldots, C_{n-1}$ cooperate to perform the following task. Each child process C_i generates its own contribution x_i (a positive integer), and sends x_i to P. Subsequently, P combines these contributions to a positive integer value $y = f(x_1, x_2, \ldots, x_n)$, and sends y to all the child processes. The processes use pipes for all these communications. All child-to-parent communications (of x_i) take place using a <u>single</u> pipe called the *parent pipe*. All parent-to-child communications (of y) take place using another <u>single</u> pipe called the *child pipe*. Each such communication must use the high-level *printf*() and *scanf*() functions (instead of *read*() and *write*()). The processes also interact with the user using the terminal. The parent reads n from the user. Each child process prints its contribution x_i to the terminal. This is also echoed by the parent process. After y is computed, P prints it, followed by all the child processes. A sample transcript that the user sees on the terminal is given to the right. The printing sequence must be exactly as illustrated in the example. Use the system call *dup*() (no other primitive is allowed) for all redirections.

Assume that each x_i is generated by a function *mycontrib*(). The parent stores the x_i values in an array `x[]`. The parent computes y by calling *combine*(x,n). You do not need to write these two functions. The codes for both P and each C_i are written in the same source file. After each fork, the new child calls a function *childmain*() which does not return. Fill out the details of this implementation on the next page. <u>Use only C constructs</u>. Follow the instructions given as comments.

```
Child(0) : x[0] = 7
Parent   : x[0] = 7
Child(1) : x[1] = 9
Parent   : x[1] = 9
Child(2) : x[2] = 4
Parent   : x[2] = 4
Child(3) : x[3] = 7
Parent   : x[3] = 7
Child(4) : x[4] = 6
Parent   : x[4] = 6
Child(5) : x[5] = 8
Parent   : x[5] = 8
Child(6) : x[6] = 4
Parent   : x[6] = 4
Child(7) : x[7] = 9
Parent   : x[7] = 9
Child(8) : x[8] = 4
Parent   : x[8] = 4
Child(9) : x[9] = 8
Parent   : x[9] = 8
Parent   : y = 359
Child(0) : y = 359
Child(1) : y = 359
Child(2) : y = 359
Child(3) : y = 359
Child(4) : y = 359
Child(5) : y = 359
Child(6) : y = 359
Child(7) : y = 359
Child(8) : y = 359
Child(9) : y = 359
```

(a) First, write the *main*() function of the parent *P*. **[7]**

```
int main ( )
{
   int n, i, x[MAX_SIZE], y;
   int pfd[2], cfd[2];                     /* variables for parent pipe and child pipe */

   printf("Enter number of child processes: "); scanf("%d", &n);

   /* Create the parent pipe and the child pipe */

   pipe(pfd);
   pipe(cfd);

   /* One by one, create child processes, scanf their contributions (via parent pipe), and printf these
      contributions to terminal. Each child jumps to childmain() with the minimal set of arguments. */

   close(0); dup(pfd[0]);

   for (i=0; ,i<n; ++i) {
      if (fork()) {
         scanf("%d", x + i);
         printf("Parent   : x[%d] = %d\n", i, x[i]);
      } else
         childmain(i,pfd[1],cfd[0]);
   }

   y = combine(x,n);
   printf("Parent   : y = %d\n", y);              /* Print to terminal */

   /* One by one printf y to all child processes (via child pipe), and wait for them to print and exit */

   close(1); dup(cfd[1]);

   for (i=0; ,i<n; ++i) {
      printf("%d\n", y); fflush(stdout);
      wait(NULL);
   }

   exit(0);
}
```

(b) Then, write the function *childmain*() for each child process. **[7]**

```
void childmain ( int i, int pfd, int cfd )
{
   /* Declare local variables */

   int x, y, fd1cpy;

   x = mycontrib();
   printf("Child(%d) : x[%d] = %d\n", i, i, x);    /* Print to terminal */

   /* Send x to parent (via parent pipe) using printf */

   fd1cpy = dup(1);        /* save original stdout for future screen printing */
   close(1); dup(pfd);
   printf("%d\n", x);

   /* Receive y from parent (via child pipe) using scanf */

   close(0); dup(cfd);
   scanf("%d", &y);
   close(1); dup(fd1cpy);  /* restore original stdout for screen printing */

   printf("Child(%d) : y = %d\n", i,  y);          /* Print to terminal */
   exit(0);                                        /* Child does not return to main() */
}
```