CS39002 Operating Systems Laboratory, Spring 2023–2024

Lab Test 1

20–March–2024 | 03:15pm–04:45pm | Maximum marks: 50

Roll no: ____________________ **Name:** ____

[*Write in the respective spaces provided. All programs are assumed to* **`#include`** *appropriate header files.*]

1. Consider the C program shown to the right. The user supplies a non-negative value of n. Prove or disprove: The number of lines printed by the program is the Fibonacci number F_{n+1}. (Recall that the Fibonacci sequence starts with $F_0 = 0$ and $F_1 = 1$.) **(10)**

```
void f ( int n )
{
   if (n < 2) {
      printf("Hi...\n");
      exit(0);
   } else {
      if (!fork()) f(n-1);
      if (!fork()) f(n-2);
   }
}
```

```
int main ( int argc, char *argv[] )
{
   int n;

   if (argc == 1) exit(1);
   n = atoi(argv[1]);
   f(n);
   exit(0);
}
```

Solution *False.* The claim is true only for $n = 0, 1, 2$. Take $n = 3$. The call $f(3)$ creates two child processes. The first child calls $f(2)$, and eventually returns and forks another process to run $f(1)$. It follows that four lines are printed for $n = 3$, whereas $F_4 = 3$. Indeed, for all $n \geqslant 3$, the number of lines printed is more than F_{n+1}.

2. (a) You compile and run the adjacent C program in a terminal (in the foreground). Two processes are created. The parent process keeps on printing **`P`**, and the child process keeps on printing **`C`**. What happens if you type control-c in the terminal? **(2)**

Solution Both the parent and the child processes terminate.

```
int main ()
{
   if (fork()) {
      while (1) {
         printf("P");
         fflush(stdout);
         sleep(1);
      }
   } else {
      while (1) {
         printf("C");
         fflush(stdout);
         sleep(1);
      }
   }
}
```

(b) Rewrite the program so that typing control-c in the terminal for the first time terminates the child process (but the parent process continues to print **P**), and typing control-c in the terminal for the second time terminates the parent process. Use appropriate signal handlers, and note that a signal handler can be redefined inside a signal handler. **(8)**

Solution The child uses no SIGINT handler. The parent first uses the handler **firstint**, and then **secondint**.

```
void secondint ( int sig )
{
   printf("\n");
   wait(NULL);
   exit(1);
}

void firstint ( int sig )
{
   signal(SIGINT, secondint);
   printf("\n");
}
```

```
int main ()
{
   if (fork()) {
      signal(SIGINT, firstint);
      while (1) {
         printf("P");
         fflush(stdout);
         sleep(1);
      }
   } else {
      while (1) {
         printf("C");
         fflush(stdout);
         sleep(1);
      }
   }
}
```

3. The Unix command **fortune** (without any argument) prints a random quotation (from a famous personality or from the fortune teller). Write a C program to do the following task. It enters a loop in which it keeps on printing the prompt **$** to the screen and reading user inputs. If the user enters **s**, then a fortune is printed to the screen. If the user enters **f**, then a fortune is printed to the file **fortunes.txt**. If the user enters anything else, the loop is broken, and the program terminates. If the file **fortunes.txt** exists before running the program, it is overwritten. All the fortunes printed by the **f** directive in a run of the program are written one after another in **fortune.txt**. You must use some **exec** function (to run **fortune** without any command-line argument) and **dup** (for suitable redirections of **stdout**). You are forbidden to use **system** or pipes. **(10)**

Solution The **main** function is given below.

```
int main ()
{
   FILE *fp;
   char resp;
   int stdoutcpy, filecpy;

   stdoutcpy = dup(1);
   fp = (FILE *)fopen("fortunes.txt", "w");
   filecpy = fileno(fp);
   while (1) {
      printf("$ ");
      scanf("%c", &resp);
      while (getchar() != '\n') ;
      if (resp == 'f') {
         close(1);
         dup(filecpy);
      } else if (resp != 's') {
         break;
      }
      if (!fork()) execlp("fortune", "fortune", NULL);
      wait(NULL);
      printf("\n");
      close(1);
      dup(stdoutcpy);
   }
   fclose(fp);
   exit(0);
}
```

4. Two programs **`first.c`** and **`second.c`** work as explained below. You compile the programs to the executable files named **`first`** and **`second`**, respectively. You first run **`first`** in a terminal, and then **`second`** in another terminal. No matter how much later you run **`second`** than **`first`**, the program **`second`** will first print **`"Hi from second"`** in its terminal, and then **`first`** will print **`"Hi from first"`** in its terminal. After the respective printing, each program will terminate. Write these two C programs. Use a semaphore for the synchronization. No other synchronization method will deserve any credit. No need to write the **`#include`**'s. **(5 + 5)**

first.c

```
int main ()
{
   int semid;
   key_t skey;
   struct sembuf P;

   skey = ftok("/home/", 'A');
   semid = semget(skey, 1, 0777 | IPC_CREAT);
   semctl(semid, 0, SETVAL, 0);
   P.sem_num = 0; P.sem_flg = 0; P.sem_op = -1;
   semop(semid, &P, 1);
   printf("Hi from first\n");
   semctl(semid, 0, IPC_RMID, 0);
   exit(0);
}
```

second.c

```
int main ()
{
   int semid;
   key_t skey;
   struct sembuf V;

   skey = ftok("/home/", 'A');
   semid = semget(skey, 1, 0777 | IPC_CREAT);
   V.sem_num = 0; V.sem_flg = 0; V.sem_op = 1;
   semop(semid, &V, 1);
   printf("Hi from second\n");
   exit(0);
}
```

5. The two C programs given below use the pthread API. Each of the two programs is meant for carrying out the task explained now. Let M be the main thread (the thread that runs **`main()`**), and T the other thread (the thread that runs **`tmain()`**). T is supposed to read the shared variable n from the user. After that, M is supposed to print the value of n that the user enters. Both the programs *may* encounter some problems that will not let them accomplish their desired tasks.

First program	Second program

<pre>pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
int n;
 	
void *tmain (void *arg)
{
   pthread_mutex_lock(&mtx);
   printf("T: n = "); scanf("%d", &n);
   pthread_mutex_unlock(&mtx);
   pthread_exit(NULL);
}
 	
int main ()
{
   pthread_t t;
 	
   pthread_create(&t, NULL, tmain, NULL);
   pthread_mutex_lock(&mtx);
   printf("M: n = %d\n", n);
   pthread_mutex_unlock(&mtx);
   pthread_exit(NULL);
}</pre>
	<pre>pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
	pthread_cond_t cnd = PTHREAD_COND_INITIALIZER;
	int n;
	
	void *tmain (void *arg)
	{
	   pthread_mutex_lock(&mtx);
	   printf("T: n = "); scanf("%d", &n);
	   pthread_cond_signal(&cnd);
	   pthread_mutex_unlock(&mtx);
	   pthread_exit(NULL);
	}
	
	int main ()
	{
	   pthread_t t;
	
	   pthread_create(&t, NULL, tmain, NULL);
	   pthread_mutex_lock(&mtx);
	   pthread_cond_wait(&cnd, &mtx);
	   pthread_mutex_unlock(&mtx);
	   printf("M: n = %d\n", n);
	   pthread_exit(NULL);
	}</pre>

(a) From the perspective of the user, what is the problem that the first program may face? **(2)**

Solution M prints an uninitialized value of n.

(b) Why does the first program face the problem that you identified in Part (a)? **(2)**

Solution If M locks **`mtx`** first, then T does not get a chance to read n from the user until M releases **`mtx`**.

(c) From the perspective of the user, what is the problem that the second program may face? **(2)**

Solution The program (M to be precise) hangs.

(d) Why does the second program face the problem that you identified in Part (c)? **(2)**

Solution T locks **`mtx`** first, and reads n as intended. T then sends a signal on the condition variable **`cnd`** before eventually releasing **`mtx`**. After all these, M gets the opportunity to lock **`mtx`** and wait on **`cnd`**. But by that time, the signal on **`cnd`** sent by T is lost.

(e) How can you write a correct program (using pthread synchronization primitives only) so that the two threads do their intended tasks? You do not need to write a program. Specify in words what the threads should do. Busy waits and **`sleep()`** (or similar calls) must not be used for synchronization. **(2)**

Solution A barrier B may be used. M initializes B to 2, and then creates T. Both M and T wait on B. M waits before printing n, whereas T waits after reading n from the user.