**PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ**
**FACULTAD DE CIENCIAS E INGENIERÍA**

**SISTEMAS OPERATIVOS**
**1ra práctica (tipo a)**
**(Primer semestre de 2019)**

Horario 0781: prof. V. Khlebnikov
Horario 0782: prof. A. Bello R.

Duración: 1 h. 50 min.
Nota: No se puede usar ningún material de consulta.
**La presentación, la ortografía y la gramática influirán en la calificación.**
Puntaje total: 20 puntos

---

**Pregunta 1 (10 puntos – 50 min.)** Analice el siguiente programa. Tome en cuenta que en el lenguaje de programación la evaluación de expresiones booleanas por defecto es de corto circuito (*short-circuit boolean-expression evaluation*), o sea, si el valor de la expresión ya está determinado, entonces el resto de la expresión no se evalúa. También recuerde que la negación, por ejemplo, de 4 es 0; la negación de 0 es 1, y que 7 nunca es igual a 8. Considerando la evaluación de la expresión booleana no olvide que el proceso hijo hereda el resultado de la parte ya evaluada por su padre.
¿Cuántos procesos hijos tendrá el primer proceso creado durante la ejecución de este programa (**2 puntos**)?
Presente la evaluación de la expresión booleana de la línea 18 del código en cada proceso hijo del primer proceso creado durante la ejecución de este programa (**2 puntos**).
¿Cuántos procesos hijos tendrá el primer hijo del primer proceso creado durante la ejecución de este programa (**2 puntos**)?
Complete los valores de los *ppid*s (marcados con "....") en la salida generada por la ejecución del programa (**4 puntos**).

```
$ cat -n kill_bill.c
     1  #include <stdio.h>
     2  #include <stdlib.h>
     3  #include <unistd.h>
     4  #include <sys/types.h>
     5  #include <sys/wait.h>
     6  #include <signal.h>
     7
     8  int main(void)
     9  {
    10    pid_t great_grandfather, ppid, bill, bill_vol2;
    11    int status, pipefd[2];
    12
    13    pipe(pipefd);
    14    dup2(pipefd[0],STDIN_FILENO); dup2(pipefd[1],STDOUT_FILENO);
    15    close(pipefd[0]); close(pipefd[1]);
    16    fprintf(stderr, "pid = %ld, ppid = %ld.\n",
    17            (long)(great_grandfather=getpid()), (long)getppid());
    18    if ((!fork() || fork()) && (fork() == !(bill=fork()))) {
    19      fprintf(stderr, "pid = %ld, ppid = %ld: true and bill = %ld\n",
    20              (long)getpid(), (long)(ppid=getppid()), (long)bill);
    21      sleep(1);
    22      if (ppid != great_grandfather) {
    23        write(1, &bill, sizeof(ppid));
    24        dup2(STDERR_FILENO,STDOUT_FILENO);
    25        execl("/bin/ps", "ps", "-l", NULL);
    26      } else {
    27        bill_vol2 = bill;
    28        read(0, &bill, sizeof(ppid));
    29        fprintf(stderr, "pid = %ld, ppid = %ld: father of %ld did ps\n",
    30                (long)getpid(), (long)getppid(), (long)bill);
    31        kill(bill, SIGTERM);
    32        fprintf(stderr, "pid = %ld: terminated.\n", (long)bill);
    33        kill(bill_vol2, SIGTERM);
    34        fprintf(stderr, "pid = %ld: terminated.\n", (long)bill_vol2);
    35        fprintf(stderr, "pid = %ld, ppid = %ld: finished.\n",
    36                (long)getpid(), (long)getppid());
    37        exit(0);
    38      }
    39    }
    40    close(0); close(1);
    41    sleep(3);
    42    while (waitpid(-1, &status, 0) != -1);
    43    fprintf(stderr, "pid = %ld, ppid = %ld: finished.\n",
    44            (long)getpid(), (long)getppid());
```

```
    45    return 0;
    46  }
$ gcc kill_bill.c -o kill_bill
$ ./kill_bill
pid = 9076, ppid = 4592.
pid = 9079, ppid = ....: true and bill = 9083
pid = 9080, ppid = ....: true and bill = 9084
pid = 9080, ppid = ....: father of 9083 did ps
pid = 9083: terminated.
pid = 9084: terminated.
pid = 9080, ppid = ....: finished.
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  4592  4585  0  80   0 -  6360 wait   pts/0    00:00:00 bash
0 S  1000  9076  4592  0  80   0 -  1095 hrtime pts/0    00:00:00 kill_bill
1 S  1000  9077  ....  0  80   0 -  1095 hrtime pts/0    00:00:00 kill_bill
1 S  1000  9078  ....  0  80   0 -  1095 hrtime pts/0    00:00:00 kill_bill
4 R  1000  9079  ....  0  80   0 -  7877 -      pts/0    00:00:00 ps
1 Z  1000  9080  ....  0  80   0 -     0 -      pts/0    00:00:00 kill <defunct>
1 S  1000  9081  ....  0  80   0 -  1095 hrtime pts/0    00:00:00 kill_bill
1 S  1000  9082  ....  0  80   0 -  1095 hrtime pts/0    00:00:00 kill_bill
1 Z  1000  9083  ....  0  80   0 -     0 -      pts/0    00:00:00 kill <defunct>
pid = 9078, ppid = ....: finished.
pid = 9081, ppid = ....: finished.
pid = 9082, ppid = ....: finished.
pid = 9077, ppid = ....: finished.
pid = 9076, ppid = 4592: finished.
```

**Pregunta 2 (2 puntos – 10 min.)** Lea el manual de `vfork(2)` (ver anexo) y considere el siguiente programa:

```
 1  /*********             testvfork.c        ********/
 2  #include <stdio.h>
 3  #include <unistd.h>
 4  #include <stdlib.h>
 5
 6  int glob=6;
 7
 8  int main(void){
 9      int var;
10      pid_t pid;
11      var = 88;
12
13      pid = vfork();
14      if( pid < 0) {
15           printf("fork failed\n");
16           exit(EXIT_FAILURE);
17      } else if ( pid != 0){
18                  printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
19                  exit(EXIT_SUCCESS);
20           } else {
21                  glob++;
22                  var++;
23                sleep(10);
24                  printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
25                _exit(EXIT_SUCCESS);
26           }
27  }
```

Después de compilar el programa, se ejecuta en la línea de ordenes:

**`./testvfork`**

Si el *pid* del padre es `20544` y el *pid* del hijo es `20545`. ¿Cuál es la salida exacta del programa? ¿Qué características de **`vfork()`** verifica el programa?

**Pregunta 3 (3 puntos – 15 min.)** La función `atexit(3)` de la librería estándar de C (ver anexo) permite registrar funciones para que sean ejecutadas cuando un proceso termina de forma normal. Las funciones no pueden tener argumentos. Se tiene el siguiente programa:

```
 1  /***********        testatexit       ***********/
 2  #include <stdio.h>
 3  #include <unistd.h>
 4  #include <stdlib.h>
 5
 6  int glob=6;
 7  static void myexit1(void), myexit2(void);
 8
 9  int main(void){
```

```
10      int var;
11      pid_t pid;
12      var = 88;
13      printf("before fork\n");
14
15      if(atexit(myexit2) != 0) {
16                printf("ERROR can't register myexit1"); exit(EXIT_FAILURE);}
17      if(atexit(myexit1) != 0) {
18                printf("ERROR can't register myexit1"); exit(EXIT_FAILURE);}
19
20      pid = fork();
21      if( pid < 0){
22            printf("fork failed\n"); exit(EXIT_FAILURE);
23       } else if ( pid == 0){
24            glob++;
25            var++;
26      }
27      printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
28      exit(EXIT_SUCCESS);
29  }
30  static void myexit2(void){
31      printf("call to myexit2, second exit handler\n");
32  }
33  static void myexit1(void){
34      printf("call to myexit1, first exit handler\n");
35  }
```

Se compila y se ejecuta la siguiente orden:

```
./testatexit > salida
cat salida
```

Si el *pid* del padre es `9159` y el *pid* del hijo es `9160`. ¿Cuál es la salida del programa **`cat`**?

**<u>Pregunta 4</u> (5 puntos – 25 min.)** El anillo de procesos . . . una vez más:

```
 1  #include <errno.h>
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <string.h>
 5  #include <unistd.h>
 6  int main(int argc,  char *argv[ ]) {
 7     pid_t childpid;             /* indicates process should spawn another     */
 8     int error;                  /* return value from dup2 call                */
 9     int fd[2];                  /* file descriptors returned by pipe          */
10     int i;                      /* number of this process (starting with 1)   */
11     int nprocs;                 /* total number of processes in ring          */
12             /* check command line for a valid number of processes to generate */
13     if ( (argc != 2) || ((nprocs = atoi (argv[1])) <= 0) ) {
14         fprintf (stderr, "Usage: %s nprocs\n", argv[0]);
15         return 1;
16     }
17     if (pipe (fd) == -1) {      /* connect std input to std output via a pipe */
18        perror("Failed to create starting pipe");
19        return 1;
20     }
21     if ((dup2(fd[0], STDIN_FILENO) == -1) ||(dup2(fd[1], STDOUT_FILENO) == -1)) {
22        perror("Failed to connect pipe");
23        return 1;
24     }
25     if ((close(fd[0]) == -1) || (close(fd[1]) == -1)) {
26        perror("Failed to close extra descriptors");
27        return 1;
28     }
29     for (i = 1; i < nprocs;  i++) {         /* create the remaining processes */
30        if (pipe (fd) == -1) {
31           fprintf(stderr, "[%ld]:failed to create \
                             pipe %d: %s\n", (long)getpid(), i, strerror(errno));
32           return 1;
33        }
34        if ((childpid = fork()) == -1) {
35           fprintf(stderr, "[%ld]:failed to create \
                              child %d: %s\n", (long)getpid(), i, strerror(errno));
36           return 1;
37        }
38        if (childpid > 0)               /* for parent process, reassign stdout */
39            error = dup2(fd[1], STDOUT_FILENO);
40        else                             /* for child process, reassign stdin */
41            error = dup2(fd[0], STDIN_FILENO);
```

```
42        if (error == -1) {
43           fprintf(stderr, "[%ld]:failed to dup pipes for \
                           iteration %d: %s\n", (long)getpid(), i, strerror(errno));
44           return 1;
45        }
46        if ((close(fd[0]) == -1) || (close(fd[1]) == -1)) {
47           fprintf(stderr,"[%ld]:failed to close extra\
                            descriptors %d:%s\n",(long)getpid(),i,strerror(errno));
48           return 1;
49        }
50        if (childpid) break;
51     }                                                  /* say hello to the world */
52     fprintf(stderr, "This is process %d with ID\
                       %ld and parent id %ld\n",i, (long)getpid(), (long)getppid());
53     return 0;
54  }
```

**(a) (1 punto – 5 min.)** Si se modifica el programa de modo que los hijos hagan **`break`** en lugar del padre, se forma un abanico de procesos. ¿Qué otros cambios hay que hacer para mantener la comunicación entre los procesos (topología)?

**(b) (1 punto – 5 min.)** Si se modifica el programa de modo que ningún proceso haga **`break`**, se forma un abanico de procesos. ¿Qué otros cambios hay que hacer para mantener la comunicación entre los procesos (topología)?

**(c) (3 puntos – 15 min.)** ¿Qué código se debe agregar si se desea emplear el anillo de procesos para calcular el número de Fibonacci. Por ejemplo:

```
./anillo 0
0
./anillo 6
8
./anillo 78
8944394323791464
```

La práctica ha sido preparada por AB (2-4) y VK (1)
con LibreOffice Writer en Linux Mint 19.1 Tessa.

Profesores del curso: (0781) V. Khlebnikov
(0782) A. Bello R.

Pando, 12 de abril de 2019

# ANEXO

```
VFORK(2)                                   Linux Programmer's Manual                                   VFORK(2)

NAME
       vfork - create a child process and block parent

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t vfork(void);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       vfork():
           Since glibc 2.12:
               (_XOPEN_SOURCE >= 500) && ! (_POSIX_C_SOURCE >= 200809L)
                   || /* Since glibc 2.19: */ _DEFAULT_SOURCE
                   || /* Glibc versions <= 2.19: */ _BSD_SOURCE
           Before glibc 2.12:
               _BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION
   Standard description
       (From  POSIX.1)  The  vfork()  function has the same effect as fork(2), except that the behavior is undefined if the process created by vfork() either modifies any data other than a
       variable of type pid_t used to store the return value from vfork(), or returns from the function in which vfork() was called, or calls any other function before successfully calling
       _exit(2) or one of the exec(3) family of functions.

   Linux description
       vfork(), just like fork(2), creates a child process of the calling process.  For details and return value and errors, see fork(2).

       vfork() is a special case of clone(2).  It is used to create new processes without copying the page tables of the parent process.  It may be useful in performance-sensitive applica-
       tions where a child is created which then immediately issues an execve(2).

       vfork() differs from fork(2) in that the calling thread is suspended until the child terminates (either normally, by calling _exit(2), or abnormally, after delivery of a fatal  sig-
       nal),  or  it makes a call to execve(2).  Until that point, the child shares all memory with its parent, including the stack.  The child must not return from the current function or
       call exit(3) (which would have the effect of calling exit handlers established by the parent process and flushing the parent's stdio(3) buffers), but may call _exit(2).

       As with fork(2), the child process created by vfork() inherits copies of various of the caller's process attributes (e.g., file descriptors, signal dispositions, and current working
       directory); the vfork() call differs only in the treatment of the virtual address space, as described above.

       Signals sent to the parent arrive after the child releases the parent's memory (i.e., after the child terminates or calls execve(2)).
```

```
ATEXIT(3)                                  Linux Programmer's Manual                                  ATEXIT(3)

NAME
       atexit - register a function to be called at normal process termination

SYNOPSIS
       #include <stdlib.h>

       int atexit(void (*function)(void));

DESCRIPTION
       The  atexit()  function registers the given function to be called at normal process termination, either via exit(3) or via return from the program's main().  Functions so registered
       are called in the reverse order of their registration; no arguments are passed.

       The same function may be registered multiple times: it is called once for each registration.

       POSIX.1 requires that an implementation allow at least ATEXIT_MAX (32) such functions to be registered.  The actual limit supported  by  an  implementation  can  be  obtained  using
       sysconf(3).

       When  a  child  process  is  created  via  fork(2),  it inherits copies of its parent's registrations.  Upon a successful call to one of the exec(3) functions, all registrations are
       removed.

RETURN VALUE
       The atexit() function returns the value 0 if successful; otherwise it returns a nonzero value.
```