PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

SISTEMAS OPERATIVOS
Examen 1
(Primer semestre de 2020)

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

Duración: 3 horas
Nota: No se puede usar ningún material de consulta.
Puntaje total: 20 puntos

Pregunta 1 (5 puntos – 30 min.)

El archivo de su respuesta debe estar en el Campus Virtual, en la carpeta de Documentos del curso: Exámenes | Examen 1 | Pregunta 1 | 0781/0782) **antes de las 08:45**. Por cada 3 minutos de retardo son -2 puntos.
El nombre de su archivo debe ser <*su_código_de_8_dígitos*>`_11.txt`. Por ejemplo, `20202912_11.txt`.

`$`

El *shell* está activo y se pregunta el valor de su PID que está en la variable del ambiente con el nombre "`$`":

```
$ echo $$
3208
```

Como en sistemas Unix casi todo son archivos, entonces los descriptores (`fd`) de los archivos abiertos por un proceso también son archivos (¡descriptores de archivos son archivos!) y los podemos visualizar en el proceso de nuestro *shell*:

```
$ ls -la /proc/$$/fd
total 0
dr-x------ 2 vk vk  0 jun  1 20:36 .
dr-xr-xr-x 9 vk vk  0 jun  1 14:42 ..
lrwx------ 1 vk vk 64 jun  1 20:36 0 -> /dev/pts/0
lrwx------ 1 vk vk 64 jun  1 20:36 1 -> /dev/pts/0
lrwx------ 1 vk vk 64 jun  1 20:36 2 -> /dev/pts/0
lrwx------ 1 vk vk 64 jun  1 20:36 255 -> /dev/pts/0
```

Y nosotros sabemos que los 3 archivos estándar siempre están abiertos:

```
$ ls -l /dev/std*
lrwxrwxrwx 1 root root 15 may 18 14:31 /dev/stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root 15 may 18 14:31 /dev/stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root 15 may 18 14:31 /dev/stdout -> /proc/self/fd/1
```

Nuestro *shell* es `bash`:

```
$ echo $SHELL
/bin/bash
```

Y parece él usa un truco guardando en el descriptor 255 el dispositivo inicial:

```
...
lrwx------ 1 vk vk 64 jun  1 20:36 255 -> /dev/pts/0
```

Lo ignoraremos por ahora.

El *shell* puede construir una tubería para recibir el flujo de *stdout* producido por `ls` y pasarlo como *stdin* a `cat`:

```
$ ls -la /proc/$$/fd | cat
total 0
dr-x------ 2 vk vk  0 jun  1 20:36 .
dr-xr-xr-x 9 vk vk  0 jun  1 14:42 ..
lrwx------ 1 vk vk 64 jun  1 20:36 0 -> /dev/pts/0
lrwx------ 1 vk vk 64 jun  1 20:36 1 -> /dev/pts/0
lrwx------ 1 vk vk 64 jun  1 20:36 2 -> /dev/pts/0
lrwx------ 1 vk vk 64 jun  1 20:36 255 -> /dev/pts/0
```

Pero lo mismo podremos hacer usando la sustitución de proceso que permite que la salida de un proceso pudiera referirse por un nombre de archivo. Este nombre de archivo pasa como un parámetro a `cat` en nuestro caso y `cat` abre este archivo y lo lee por tubería:

```
$ cat <(ls -la /proc/$$/fd)
total 0
dr-x------ 2 vk vk  0 may 31 20:58 .
dr-xr-xr-x 9 vk vk  0 may 23 12:28 ..
lrwx------ 1 vk vk 64 may 31 20:58 0 -> /dev/pts/4
lrwx------ 1 vk vk 64 may 31 20:58 1 -> /dev/pts/4
lrwx------ 1 vk vk 64 may 31 20:58 2 -> /dev/pts/4
lrwx------ 1 vk vk 64 may 31 20:58 255 -> /dev/pts/4
lr-x------ 1 vk vk 64 jun  1 14:09 63 -> pipe:[11218030]
```

Y el nombre de archivo creado en esta sustitución de proceso por archivo es `/dev/fd/63` porque `echo` imprime este nombre cuando lo indicamos como su argumento:

```
$ echo <(ls -la /proc/$$/fd)
/dev/fd/63
```

Hay 2 programas simples `true` y `false` que devuelven 0 (true) y 1 (false):

```
$ true
$ echo $?       # ¿cómo terminó true? Esto está todavía en la variable con el nombre "?"
0
$ false
$ echo $?       # ¿cómo terminó false?
1
```

Entonces, cuando el proceso que ejecuta `true` se sustituye por un archivo, se usa el descriptor 63, y cuando se necesita otro, se usa también el descriptor 62:

```
$ echo >(true)
/dev/fd/63
$ echo <(true)
/dev/fd/63
$ echo <(false) >(true)
/dev/fd/63 /dev/fd/62
```

Escribimos un programa para imprimir los descriptores de archivos usados:

```
$ cat -n fds_print_ps.c | expand
     1  /*
     2      https://stackoverflow.com/questions/37396241/printing-file-descriptors-from-pid-in-c-on-linux
     3  */
     4
     5  #include <stdio.h>
     6  #include <fcntl.h>
     7  #include <stdlib.h>
     8  #include <error.h>
     9  #include <dirent.h>
    10  #include <string.h>
    11  #include <unistd.h>
    12
    13  int
    14  main(int argc, char *argv[])
    15  {
    16      char *path = "/proc/self/fd";
    17      DIR *fd_dir = opendir(path);
    18
    19      if (!fd_dir) {
    20          perror("opendir");
    21          exit(EXIT_FAILURE);
    22      }
    23
    24      printf("pid: %d, ppid: %d\n", getpid(), getppid());
    25
    26      int fd;
    27      if (argc == 2) {
    28          if ((fd=open(argv[0], O_RDONLY)) < 0)
    29              perror("open");
    30          printf("fd: %d\n", fd);
    31      }
    32
    33      struct dirent *cdir;
    34      size_t fd_path_len = strlen(path) + 10;
    35
```

```
36      char *fd_path = malloc(sizeof(char)*fd_path_len);
37      char *buf = malloc(sizeof(char)*PATH_MAX + 1);
38
39      while ((cdir = readdir(fd_dir))) {
40          if (strcmp(cdir->d_name,".") == 0 ||
41              strcmp(cdir->d_name,"..") == 0)
42              continue;
43          snprintf(fd_path, fd_path_len-1, "%s/%s", path, cdir->d_name);
44          printf("Checking: %s: ", fd_path);
45
46          ssize_t link_size = readlink(fd_path, buf, PATH_MAX);
47          if (link_size < 0)
48              perror("readlink");
49          else {
50              buf[link_size] = '\0';
51              printf("%s\n", buf);
52          }
53          memset(fd_path, '0', fd_path_len);
54      }
55      if (argc == 3)
56          if (system("ps -l") < 0)
57              perror("system");
58
59      closedir(fd_dir);
60      if (argc == 2) close(fd);
61      free(fd_path);
62      free(buf);
63      exit(EXIT_SUCCESS);
64  }
```

```
$ gcc fds_print_ps.c -o fds_print_ps
```

```
$ ./fds_print_ps
pid: 395, ppid: 26880
pid: 725, ppid: 3208
Checking: /proc/self/fd/0: /dev/pts/0
Checking: /proc/self/fd/1: /dev/pts/0
Checking: /proc/self/fd/2: /dev/pts/0
Checking: /proc/self/fd/3: /proc/725/fd
$
```

a) (3 puntos) Se esperaba ver en la salida del programa, presentada encima de esta línea, solo 3 descriptores de archivos estándar pero aquí hay 4. ¿Por qué?

Bueno, las siguientes ejecuciones proporcionan más información:

```
$ ./fds_print_ps 1
pid: 1542, ppid: 3208
fd: 4
Checking: /proc/self/fd/0: /dev/pts/0
Checking: /proc/self/fd/1: /dev/pts/0
Checking: /proc/self/fd/2: /dev/pts/0
Checking: /proc/self/fd/3: /proc/1542/fd
Checking: /proc/self/fd/4: /home/vk/clases/so/progs/fds_print_ps
```

```
$ ./fds_print_ps 1 2
pid: 1554, ppid: 3208
Checking: /proc/self/fd/0: /dev/pts/0
Checking: /proc/self/fd/1: /dev/pts/0
Checking: /proc/self/fd/2: /dev/pts/0
Checking: /proc/self/fd/3: /proc/1554/fd
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1554  3208  0  80   0 -  1127 wait   pts/0    00:00:00 fds_print_ps
0 S  1000  1555  1554  0  80   0 -  1157 wait   pts/0    00:00:00 sh
0 R  1000  1556  1555  0  80   0 -  7878 -      pts/0    00:00:00 ps
0 S  1000  3208 21839  0  80   0 -  6617 wait   pts/0    00:00:00 bash
```

b) (1 punto) ¿Qué información proporcionan estas dos salidas? Analícelo aplicando sus conocimientos obtenidos en este curso.

Y la última ejecución:

```
$ (./fds_print_ps 1 2)> >(cat)
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  3208 21839  0  80   0 -  6617 wait   pts/0    00:00:00 bash
0 S  1000  3674  3208  0  80   0 -  1127 wait   pts/0    00:00:00 fds_print_ps
1 S  1000  3675  3674  0  80   0 -  6360 wait   pts/0    00:00:00 bash
0 S  1000  3676  3675  0  80   0 -  2532 pipe_w pts/0    00:00:00 cat
0 S  1000  3677  3674  0  80   0 -  1157 wait   pts/0    00:00:00 sh
0 R  1000  3678  3677  0  80   0 -  7878 -      pts/0    00:00:00 ps
```

```
pid: 3674, ppid: 3208
Checking: /proc/self/fd/0: /dev/pts/0
Checking: /proc/self/fd/1: pipe:[11758297]
Checking: /proc/self/fd/2: /dev/pts/0
Checking: /proc/self/fd/3: /proc/3674/fd
Checking: /proc/self/fd/63: pipe:[11758297]
$
```

c) (1 punto) Explique de manera justificada cómo se ejecutó este orden de *shell* usando al máximo la información proporcionada.

<u>Pregunta 2</u> (5 puntos – 30 min.)

El archivo de su respuesta debe estar en el Campus Virtual, en la carpeta de Documentos del curso: Exámenes | Examen 1 | Pregunta 2 | 0781/0782) **antes de las 09:30**. Por cada 3 minutos de retardo son -2 puntos.
El nombre de su archivo debe ser <*su_código_de_8_dígitos*>`_12.txt`. Por ejemplo, `20202912_12.txt`.

Se desea copiar el arreglo *a*[1..*n*] de enteros, al arreglo *b*[1..*n*] del mismo tipo, empleando *buf* como variable compartida. Se decide emplear la figura del Productor-Consumidor. Debe haber exclusión mutua en el uso de la variable *buf* y además el Productor y el Consumidor deben sincronizarse para que la copie se lleve a cabo apropiadamente.

Para implementar este programa emplearemos la instrucción *atómica* <**await** *B*> donde *B* es una condición booleana, tal como $s < 0$. El comportamiento es el siguiente: el proceso que invoca **await** hace un *delay* (posiblemente haciendo espera ocupada) hasta que *B* sea cierto. Recuerde que la instrucción es atómica.

a) (3 puntos – 18 min.) Dado el siguiente programa, determine B_1 y B_2 para que lleve a cabo lo descrito arriba:

```
var buf: int, p : int := 0, c : int := 0
Producer:: var a[1:n]: int
           do p < n → < await B1 >
                        buf := a[p + 1]
                        p :=p + 1
           od
Consumer:: var b[1:n]: int
           do c < n →  < await B2>
                        b[c + 1]:=buf
                        c := c + 1
           od
```

La instrucción atómica **await**, también puede tener la siguiente forma: <**await** $B \rightarrow I$> donde *B* es una condición booleana e *I* es una instrucción. En este caso el proceso que invoca **await** hace un *delay* (posiblemente haciendo espera ocupada) hasta que *B* sea cierto y a continuación ejecuta *I*, todo de forma atómica.

b) (2 puntos – 12 min.) Dado el siguiente programa, determine $B_1 \rightarrow I_1$ y $B_2 \rightarrow I_2$ para que lleve a cabo la misma tarea.

```
var buf: int, empty : bool := true, full: bool := false
Producer:: var a[1:n] : int,p : int := 1
           do p < n →  < await B1 → I1 >
                        buf:= a[p];full := true
                        p := p + 1
           od


Consumer:: var b[1:n] : int, c : int := 1
           do c < n → < await B2 → I2 >
                        b[c] := buf ; empty := true
                        c := c + 1
           od
```

Pregunta 3 (5 puntos – 30 min.)

El archivo de su respuesta debe estar en el Campus Virtual, en la carpeta de Documentos del curso: Exámenes | Examen 1 | Pregunta 3 | 0781/0782) **antes de las 10:15**. Por cada 3 minutos de retardo son -2 puntos.
El nombre de su archivo debe ser <*su_código_de_8_dígitos*>`_13.txt`. Por ejemplo, `20202912_13.txt`.

a) (4 puntos – 24 min.) Analice el siguiente fragmento de código donde los semáforos `sem1` y `sem2` están inicializados a cero, un hilo ejecuta la función incrementa y otro la función decrementa. Describa los valores que, durante la ejecución, puede adoptar la variable `num` así como las posibles situaciones de interbloqueo que pudiera presentarse.

```
int num=10;

void * incrementa(void *nada) {
   int i;
   for (i=0;i<3;i++){
         sem_wait(&sem1);
         num++;
         printf("Inc. Numero = %d \n",num);
         sem_post(&sem1);
   }
   sem_post(&sem2);
   sleep(random() %3);
   sem_wait(&sem2);
   pthread_exit(NULL);
}

void * decrementa(void *nada){
   int i;
   for (i=0;i<3;i++){
       sem_post(&sem1);
       sleep(random() %3);
       sem_wait(&sem2);
       num--;
       printf("Dec. Numero = %d\n",num);
       sem_post(&sem2);
       sem_wait(&sem1);
   }
   sem_wait(&sem1);
   pthread_exit(NULL);
}
```

b) (1 punto – 6 min.) Se crean dos hilos de manera que uno ejecuta `escribirA` y el otro `escribirB`. Agregue dos semáforos para que la salida sea BABABABABA. No olvides indicar los valores iniciales de los semáforos que utilice.

```
void *escribirA (void *p) {
     int i;
     for (i= 0; i< 5; i++) {
         printf ("A"); fflush(NULL); sleep(random() %2);
     }
     pthread exit(NULL);
}

void *escribirB (void *p) {
     int i;
     for (i= 0;i< 5; i++) {
          printf ("B"); fflush(NULL); sleep(random() %2);
     }
     pthread exit(NULL);
}
```

ANEXO

`int sem_wait(sem_t *sem);` decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

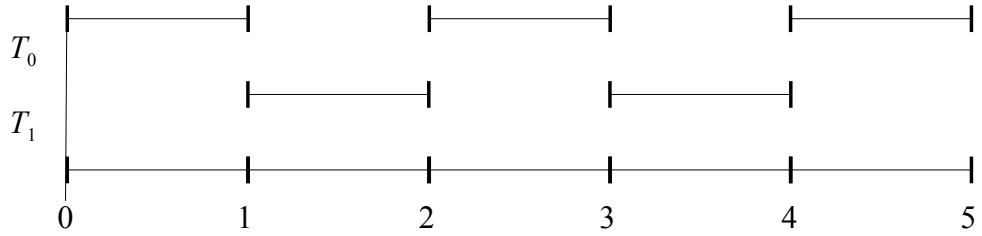
`int sem_post(sem_t *sem);` increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then an other process or thread blocked in a sem_wait(3) call will be woken up and proceed to lock the semaphore.

Pregunta 4 (5 puntos – 30 min.) El archivo de su respuesta debe estar en el Campus Virtual, en la carpeta de Documentos del curso: Exámenes | Examen 1 | Pregunta 4 | 0781/0782) **antes de las 11:00**. Por cada 3 minutos de retardo son -2 puntos. El nombre de su archivo debe ser <*su_código_de_8_dígitos*>`_14.txt`. Por ejemplo, `20202912_14.txt`.

Real-time systems are those that have requirements on their response times. For example, a flight control system is required to sample sensors and issue appropriate commands to the flight controls every t milliseconds, where t ranges from 5 to 50 milliseconds. Real-time systems are constructed by dividing up the computation into short *tasks* and then *scheduling* the tasks. The scheduling may be synchronous, where each task is given one or more slots within a period of time, or asynchronous, where the tasks are given priorities and a preemptive scheduler ensures that a lower-priority task is not run if a higher-priority task is ready.

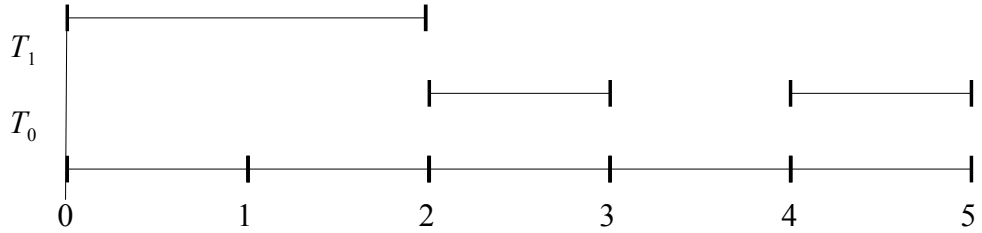
Tasks in a real-time system are generally defined to be *periodic*: with each task we associate a *period p* and an *execution time e*. The task is required to execute at least once every p units of time (microseconds or milliseconds or seconds), and it needs at most e units to complete its execution. Consider, for example, two tasks T_0 and T_1, such that $p_0 = 2$, $e_0 = 1$, and $p_1 = 5$, $e_1 = 2$; that is, T_0 needs 1 unit out of every 2 units, and T_1 needs 2 units out of every 5. We now ask if there is a *feasible* assignment of priorities, that is, if there is an assignment of priorities such that each task receives the execution time it requires when the task are scheduled by an asynchronous scheduler.

The following diagram shows that assigning T_0 a higher priority than T_1 is feasible:



T_0 receives the first unit out of every two. The execution of T_1 starts at time 1 and is interrupted at time 2 because the higher priority task T_0 is now ready to execute. Task T_1 resumes execution at time 3. In total, T_1 receives two units out of every five as required.

Not all assignments are feasible. Assigning a higher priority to T_1 results in the computation shown in the following diagram:



T_1 executes for two units without interruption, by which time T_0 has not received one unit out of the two that it needs.

It can be shown that if there is a feasible priority assignment, then *rate monotonic scheduling* is feasible. This is achieved by assigning priorities in inverse order of the *periods* of the tasks, that is, the shorter the period, the higher the priority. This is the reason that task T_0 of the example was given a higher priority.

Si su código de estudiante de esta Universidad es un número impar, analice las condiciones de factibilidad de planificación de las siguientes tareas:

$(p_1 = 8, e_1 = 3), (p_2 = 9, e_2 = 3), (p_3 = 15, e_3 = 3).$

Si su código de estudiante de esta Universidad es un número par, analice las condiciones de factibilidad de planificación de las siguientes tareas:

$(p_1 = 8, e_1 = 4), (p_2 = 12, e_2 = 4), (p_3 = 20, e_3 = 4).$

Preparado por AB (2,3) y VK (1,4)
con LibreOffice Writer en Linux Mint 19.3 "Tricia"

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

Lima, 3 de junio de 2020