PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

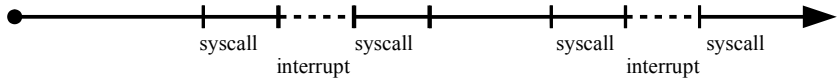
SISTEMAS OPERATIVOS
Examen 1
(Primer semestre de 2023)

Horarios 0781, 0782: prof. V. Khlebnikov

Duración: 3 horas
Notas: No está permitido el uso de ningún material o equipo electrónico.
La presentación, la ortografía y la gramática influirán en la calificación.
Puntaje total: 20 puntos

Pregunta 1 (3 puntos – 24 min.)

(a) (1 punto) Se presenta la traza de ejecución de un proceso que realiza las llamadas al sistema. La traza tiene 8 segmentos. Para cada segmento indique el estado del proceso y el modo de ejecución cuando esto es apropiado.



(b) (1 punto) ¿Cuál es la llamada al sistema que pone un proceso al estado zombi?

(c) (1 punto) En Unix, donde todo son archivos, para copiarlos se usan los programas **`cp`** y **`cat`**. Pero también se usa el programa **`dd`** que copia el contenido de un archivo bloque por bloque. Con **`dd`** se puede copiar solo una parte de archivo (usando la opción **`count`**) y se puede continuar una copia interrumpida (usando las opciones **`seek`** y **`skip`**). También se usa este programa para borrar los discos duros tomando como el archivo de entrada **`/dev/zero`**, o mejor **`/dev/urandom`**. El funcionamiento de **`dd`** parece, por ejemplo, al funcionamiento de **`cp`**:

```
$ dd if=/dev/sr0 of=~/Descargas/kill_bill_v1.iso bs=2048 count=3725646
3725646+0 registros leídos
3725646+0 registros escritos
7630123008 bytes (7,6 GB) copiados, 1942,88 s, 3,9 MB/s        # 1942 s = 32 m
$

$ dd if=/dev/urandom of=/dev/sdb1
1953456129+0 registros leídos
1953456128+0 registros escritos
1000169537536 bytes (1,0 TB) copiados, 177368 s, 5,6 MB/s      #  177368 s = 2956 m = 49 h
$
```

En algunos casos, mientras este programa se ejecuta, se usa la orden para enviar la señal **`USR1`** (no **`KILL`**) al proceso:

```
$ killall -USR1 dd
```

que produce el mismo efecto que la orden **`kill -USR1 <pid>`** pero permite identificar el proceso por el nombre del programa ejecutado por este proceso en vez de indicar su **`<pid>`** que todavía hay que averiguar.

Intente proponer para qué se usa esta última orden y qué efecto tiene. Esta propiedad de **`dd`** se menciona en su manual.

Pregunta 2 (5 puntos – 40 min.) (*AB*) Dado el siguiente código

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
#include <math.h>

double final;
char cadena[60];
void crea_procesos(int);
```

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

    n = atoi(argv[1]);
    final = pow(2, (n-1));
    crea_procesos(1);
    exit(0);
}

void crea_procesos(int x)
{
    int size;

    size = sprintf(cadena, "Proceso # %0.2d: pid=%d, ppid=%d\n", x, getpid(), getppid());
    write(1, cadena, size);
    if (x >= final) return;
    if (!fork()) { crea_procesos(2*x); exit(0); }
    if (!fork()) { crea_procesos(2*x+1); exit(0); }
    wait(NULL);
    wait(NULL);
}
```

a) (3 puntos) Presente gráficamente el árbol de procesos (indicando el # asignado a cada proceso) que se genera durante la ejecución del programa de la siguiente manera:

```
$ ./preg2 4
```

b) (2 puntos) Modifique la función `main()` de tal forma que el proceso padre pueda recoger todas las salidas de los procesos hijos y ordenarlas. La función `crea_procesos()` no debe modificarse. El proceso padre puede encargar a un proceso la ejecución del programa `sort` que se ubica por la ruta `/usr/bin/sort`.

<u>Pregunta 3</u> (4 puntos – 32 min.) Hay dos tipos de hilos – A y B, con los siguientes códigos para intercambiar sus valores:

```
    semaphore A=0, B=0;
    int       buf_A, buf_B;           /* shared variables */

 A: {                                      B: {
      int var_A; / *local variable */           int var_B; / *local variable */

1     while (1) {                         1     while (1) {
2         ...                             2         ...
3         var_A = ...;                    3         var_B = ...;
4         signal(B);                      4         signal(A);
5         wait(A);                        5         wait(B);
6         buf_A = var_A;                  6         buf_B = var_B;
7         var_A = buf_B;                  7         var_B = buf_A;
      }                                         }
     }                                         }
```

a) (2 puntos – 16 min.) Si hay una sola instancia del hilo tipo A y una sola instancia del hilo tipo B que intentan lograr un *rendezvous* para intercambiar sus valores, ¿cuál es el problema con este código? Presente la secuencia de ejecución problemática indicando las líneas ejecutadas como A3, B3, etc.

b) (2 puntos – 16 min.) Si hay dos instancias (A_1, A_2) del hilo tipo A y también dos instancias (B_1, B_2) del hilo tipo B, ¿cómo puede ocurrir que los hilos B no reciben todos los valores enviados por los hilos A? Presente la secuencia de ejecución indicando las líneas ejecutadas como $A_1$3, $B_2$3, etc.

<u>Pregunta 4</u> (4 puntos – 32 min.) Aquí están el segundo (pregunta 4a) y el tercer (pregunta 4b) intentos con el código modificado de la pregunta anterior (3b) para un intercambio de valores entre las 2 parejas de dos tipos de hilos.

a) (2 puntos – 16 min.) En el código que sigue, después que una pareja de hilos A y B completen la primera fase del intercambio de mensajes, no se sabe si ellos van a continuar con su segunda fase del intercambio u otra pareja de hilos comience con su primera fase. También es posible que uno de la pareja actual continúe e intercambia su valor con un nuevo hilo del otro grupo. Para este último caso presente una situación que provoca una *race condition*. Presente la secuencia de ejecución indicando las líneas ejecutadas como $A_1$3, $B_2$3, etc.

```
    semaphore A=0, B=0;
    semaphore mutex=1;
    int       buf_A, buf_B;           /* shared variables */
```

```
A: {                                        B: {
     int var_A; / *local variable */             int var_B; / *local variable */

1    while (1) {                            1    while (1) {
2        ...                                2        ...
3        var_A = ...;                       3        var_B = ...;
4        signal(B);                         4        signal(A);
5        wait(A);                           5        wait(B);
6        wait(mutex);                       6        wait(mutex);
7            buf_A = var_A;                 7            buf_B = var_B;
8        signal(mutex);                     8        signal(mutex);
9        signal(B);                         9        signal(A);
10       wait(A);                           10       wait(B);
11       wait(mutex);                       11       wait(mutex);
12           var_A = buf_B;                 12           var_B = buf_A;
13       signal(mutex);                     13       signal(mutex);
14   }                                      14   }
15 }                                        15 }
```

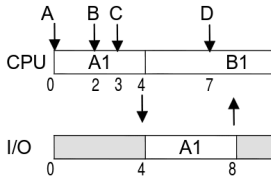
b) (2 puntos – 16 min.) Con el siguiente código es posible la perdida de valores. Presente el caso. Presente la secuencia de ejecución indicando las líneas ejecutadas como A3, B3, etc.

```
semaphore aReady=1, bReady=1;
semaphore aDone=0; bDone=0;
int       buf_A, buf_B;           /* shared variables */

A: {                                        B: {
     int var_A; / *local variable */             int var_B; / *local variable */

1    while (1) {                            1    while (1) {
2        ...                                2        ...
3        var_A = ...;                       3        var_B = ...;
4        wait(aReady);                      4        wait(bReady);
5            buf_A = var_A;                 5            buf_B = var_B;
6            signal(aDone);                 6            signal(bDone);
7            wait(bDone);                   7            wait(aDone);
8            var_A = buf_B;                 8            var_B = buf_A;
9        signal(aReady);                    9        signal(bReady);
10   }                                      10   }
11 }                                        11 }
```

<u>Pregunta 5</u> (4 puntos – 32 min.) Considere los siguientes datos y complete los diagramas de tiempo de CPU y de E/S para los siguientes algoritmos. También calcule el tiempo promedio de *turnaround* y de espera.

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1

a) FCFS



b) Shortest-Process-First (SPF) for processor scheduling and FCFS for I/O.

c) Shortest-Remaining-Time-First (SRTF) for processor scheduling.

d) Round-Robin (RR) with time quantum of 3 for processor scheduling.

Preparado por VK
con LibreOffice Writer en Linux Mint 21.1 “Vera”

Profesor del curso: V. Khlebnikov

Lima, 17 de mayo de 2023