Programación de Sistemas
Examen Final 1er Término 2019

Nombre: _____ Paralelo: ____

Pregunta 1 (30 puntos)

Considere el siguiente código multi-hilo. El programa lo ejecutamos con el siguiente comando:

./a.out 10

```c
#define FIN 1
#define NO_FIN 0

sem_t mutex;

typedef struct data{

    int cont;
    int estado;
    struct data *anterior;
    struct data *siguiente;
}Data;

int total = 0;

void *fn1(void *arg){
    Data *data = (Data *)arg;
    sem_wait(&mutex);
    data->cont += data->anterior->cont;
    data->estado = FIN;
    total += data->cont;
    sem_post(&mutex);
    return (void *)0;
}
```

```c
int main(int argc, char **argv){
    sem_init(&mutex, 0, 1);
    int n = atoi(argv[1]);
    pthread_t *ids =
    malloc(sizeof(pthread_t)*n);
    Data *ant = NULL;
    Data *primero = NULL;
    for(int i = 0; i < n; i++){
        Data *d = malloc(sizeof(Data));
        d->cont = i;
        d->estado = NO_FIN;
        if(ant == NULL){
            d->anterior = d;
            primero = d;
            d->estado = FIN;
        }else{

            d->anterior = ant;
            ant->siguiente = d;
            if(i == (n-1)){
                d->siguiente = d;
            }
        }
        ant = d;
        pthread_create(&ids[i], NULL,
                       fn1, (void *)d);
    }

    for(int i = 0; i < n; i++){
       phtread_join(ids[i], NULL);
        printf("Hilo %d -> cont %d", i,
                primero->cont);
        primero = primero->siguiente;
    }
    printf("Total %d", total);
    return 0;
}
```

¿Funciona este programa correctamente? Si el programa contiene errores, indique cuáles son, y corríjalos.

ant = d; +2 pthread_join +8

semáforos +20

Errores en morado +1 punto extra

Pregunta 2 (30 puntos)

Abajo se muestra un programa que recibe las rutas de varios archivos como argumentos, y luego cuentas las palabras para cada uno, usando un hilo por archivo. El programa tiene el problema que eventualmente se queda sin memoria. Las funciones *contar_palabras* y *procesar_linea* no tiene fugas de memoria, ni el programa principal tampoco.

```
struct conteo{
    char *palabra;
    unsigned long veces;
    struct conteo *siguiente;
}

void* procesar(void *argv)
    char **palabras = NULL;
    struct conteo= NULL;

    fd = open((char *)arg, O_RDONLY);
    char ruta[100] = {0};
    snprintf(ruta, 100,
           "%s_res.out", (char *)arg);

    while(palabras = leer_linea(fd))
          != NULL){
      contador = contarpal(contador,
                           palabras);
    }
    int p = guardar(ruta);
    free_contador(conteo);
    return (void *)p;
}
```

```
int main(int argc, char **argv){
    printf("Procesando archivos...");
    int n = argc - 1;
    long tam_pth = sizeof(pthread_t);
    pthread_t *ids = malloc(tam_pth *n);

    for(int i = 1; i < argc - 1; i++){
        pthread_create(&ids[i], NULL,
              procesar, (void *)argv[i]);
    }
    int *status;
    for(int i = 1; i < argc - 1; i++){
        pthread_join(ids[i], &status);
        if(!status)
            printf("El archivo %s se "
                "proceso OK\n", argv[i]);
        else
            printf("El archivo %s NO se "
                   "proceso. Error %d\n",
                    argv[i], *status);
    }
    free(ids);
    return 0;
}
```

1. Reescriba el programa mostrado arriba usando **PROCESOS**.

Usa fork +5
Valida que solo padre haga fork +8
Hijo llama procesar +8
Padre hace waitpid +7
Obtiene status correctamente (WEXITSTATUS) +2

Una potencial solución

```
int main(int argc, int **argv){
	printf("Procesando archivos...");
	pid_t pid_main = getpid();
	pid_t *hijos = malloc(sizeof(pid_t)*(argc-2));

	for(int i = 1; i < argc - 1; i++){
		hijos [i-1] = fork();
		if(hijos[i-1] == 0){
			long stat = (long)procesar(argv[i]);
			return stat;		//si no pusieron este return
						//necesitaban
						// if(getpid() == pid_main)
						//antes del fork
		}

	}

	int ret;
	for(int i = 1; i < argc - 1; i++){	//hijos nunca llegan aquí
	waitpid(hijos[i-1], &ret, NULL);
	 int status = WEXITSTATUS(ret);
	if(!status)
		printf("El archivo %s se proceso OK\n", argv[i]);
	else
		printf("El archivo %s NO se proceso. Error %d\n", argv[i], status);
	}
}
```

Pregunta 3 (20 puntos)

El siguiente código fue extraído de un programa donde un hilo genera de manera continua valores aleatorios **token** (hilo2) y otro hilo los inserta en un arreglo **arr** (hilo1). Debido a que ambos hilos necesitan acceder o modificar **token** y **arr**, se crearon dos semáforos binarios para proteger el acceso a ambas variables.

```
sem_t mutex_token;
sem_t mutex_arreglo;

int token = 0;
int indice = 0;
int arr[MAX];
```

```
void *hilo1(void *vargp)
{
    int cnt = 100;
    int token_temp;
    while(cnt--){
        sem_wait(&mutex_token);
        sem_wait(&mutex_arreglo);
        //sem_wait(&mutex_token);
        token_temp = arr[indice];
        if(token_temp != token){
            printf("Valor nuevo: %d\n",  token);
            printf("Valor viejo: %d\n", token_temp);
            arr[++indice%MAX] = token;
        }
        sem_post(&mutex_arreglo);
        sem_post(&mutex_token);
    }
    return NULL;
}
```

```
void *hilo2(void *vargp)
{
    int cnt = 10;
    while(cnt--){
        sem_wait(&mutex_token);
        sem_wait(&mutex_arreglo);
        token = (int) rand()%100;
        printf("Valor nuevo: %d\n",  token);
        printf("Valor viejo: %d\n", arr[indice]);
        sem_post(&mutex_token);
        sem_post(&mutex_arreglo);
    }

    return NULL;
}
```

```
int main(int argc, char **argv)
{
    sem_init(&mutex_token, 0, 1);
    sem_init(&mutex_arreglo, 0, 1);
    ...
```

a) Al ejecutar, el programa funciona normalmente en la mayoría de los casos, sin embargo, en algunas ocasiones el programa se queda congelado en plena ejecución y nunca termina. ¿Por qué sucede esto? ¿Cuál es el nombre técnico de esta situación y como se la podría arreglar?

El programa a veces se congela porque puede suceder que justamente **hilo1** este esperando en *mutex_token* e **hilo2** esperando en *mutex_arreglo* al mismo tiempo.
(Explica el problema de manera genérica 5 puntos)

Ambos hilos esperarán por siempre, esta situación se conoce como **deadlock**.
(Identifica el término correcto 5 puntos)

Se puede arreglar simplemente cambiando el orden de los sem_wait en **hilo1** para que coincidan con el orden de los sem_wait en **hilo2** (o viceversa).
(*Repara el error 5 puntos*)

b) Otra forma de arreglar este programa es re-escribiendolo completamente usando un patrón de diseño conocido, ¿cuál es este patrón de diseño?

Productor - consumidor (5 puntos)
El hilo2 sería el productor y el hilo1 el consumidor.

Pregunta 4 (20 puntos)

El archivo */proc/meminfo* contiene información de uso de la memoria del sistema, este archivo es actualizado en tiempo real por el kernel. Abajo se muestra un ejemplo del típico contenido en */proc/meminfo* y la función *main* del programa **mem.c**:

Función *main* en **mem.c**	Ejemplo contenido **/proc/meminfo**

int main(int argc, char **argv)
{
  int fd = open("/proc/meminfo", O_RDONLY, 0);
  int n, mem;
  char buffer[128];
  if(fd > 0){
  while(1){
  lseek(fd,0,SEEK_SET);
  n = read(fd, buffer, 128);
  n = sscanf(buffer, "MemTotal: %*d kB\nMemFree: %d",
&mem);
  if(n == 1){
  printf("\r%d", mem);
  fflush(stdout);
  }
  }
  }else
  fprintf(stderr, "Error: %s\n", strerror(errno));
}
	MemTotal: 4039536 kB
	MemFree: 637800 kB
	MemAvailable: 2108260 kB
	Buffers: 682056 kB
	Cached: 923528 kB
	SwapCached: 0 kB
	Active: 2177488 kB
	Inactive: 817320 kB
	Active(anon): 1282376 kB
	Inactive(anon): 209412 kB

a) ¿Qué hace el programa **mem.c**? Describa para qué podría servir este programa y proporcione un ejemplo de salida en consola.

El programa muestra en consola en tiempo real la cantidad en kB de memoria libre (MemFree).
(Explica que hace, 5 puntos)

Esto puede servir para monitorear el uso de memoria o recursos computacionales.
(Proporciona algún ejemplo de utilidad del programa, 2 puntos)

Ejemplo de salida:
> 637800
(Proporciona un ejemplo correcto de salida en consola, 3 puntos)

b) La función *lseek* tiene la siguiente descripción:

```
off_t lseek(int fd, off_t offset, int whence);
lseek() repositions the file offset of the open file
description associated with the file descriptor fd…
SEEK_SET: The file offset is set to offset bytes.
```

¿Cuál es la razón por la cual se utiliza *lseek*? ¿Qué pasaría si no se lo utiliza?

La función lseek desplaza el puntero del archivo, en este caso lo desplaza a la posición 0, es decir, al inicio del archivo.
(Explica que hace lseek en este programa, 5 puntos)

Si no se utiliza lseek para regresar el puntero de archivo a la posición inicial, el valor de MemFree sería encontrado tan solo en la primera iteración del lazo. Finalmente, el puntero de archivo llegaría al final del archivo después de algunas iteraciones y el programa no mostraría nunca valores actualizados de MemFree.
(Explica qué pasaría si no se utiliza la función, 5 puntos)