Pregunta 1

Parte a.- (3 puntos) Ud. dispone de la función *integral* del control 2:

```
typedef double (*Funcion)(void *ptr, double x);
double integral(Funcion f, void *ptr,
                double xi, double xf, int n);
```

Recuerde que $integral(f, ptr, xi, xf, n)$ calcula $\int_{xi}^{xf} f(ptr, x)dx$ numéricamente. Programe la función:

```
double integral_par(Funcion f, void *ptr,
                    double xi, double xf, int n, int k);
```

Esta función debe calcular la integral usando *k* cores en paralelo.

Restricción: Ud. debe usar *fork* para crear procesos pesados (*no threads*). Cada uno de estos procesos calcula la integral de un subintervalo y devuelve el resultado al proceso padre usando un *pipe*.

Parte b.- (2 puntos) Considere la siguiente función:

```
typedef void (*VoidFun)(void *ptr);
int proteger( VoidFun fun, void *ptr) {
  (*fun)(ptr);
  return 0;
}
```

en donde el tipo *VoidFun* corresponde al tipo de las funciones que reciben un puntero opaco (*void**) y no retornan nada. Modifique la función *proteger* de tal forma que si durante la ejecución de *fun* se produce un *segmentation fault* (señal SIGSEGV) o una división por 0 (señal SIGFPE), entonces *proteger* retorna 1, en vez de terminar el proceso que es lo usual.

Parte c.- (1 punto) Programe la siguiente función:

```
int integral_prot(Funcion f, void *ptr,
                  double xi, double xf, int n, double *res);
```

Esta función realiza el mismo cálculo que la función *integral*, entregando el resultado en **res* y retornando 0. Pero si durante la invocación de *f* se produce un *segmentation fault* o una división por 0, *integral_prot* retorna 1 en vez de terminar el proceso.

Restricción: Ud. debe usar la función *proteger* para programar esta función.

Pregunta 2

Parte i.- (4,5 puntos) Considere el problema de la cena de filósofos. Los filósofos no saben programar pero sí saben usar el shell de comandos. Por lo tanto usarán el comando *palitos* para pedir permiso antes de comenzar a comer su plato de arroz y para notificar que terminaron de comer. Ud. debe programar (1) el comando *palitos* (el cliente) , y (2) el comando *restaurant* (el servidor) con el cual se comunica el comando palitos. Todos los comandos se ejecutarán en la misma máquina *localhost*. El servidor escucha los requerimientos de conexión en el *port* 3000. El siguiente ejemplo de uso muestra qué parámetros recibe el comando palitos y cómo se debe comportar:

Filósofo 0	*Filósofo 1*	*Filósofo 2*	*Filósofo 3*
`% ./palitos e 0` `%`			
	`% ./palitos e 1` *(espera)*		
			`% ./palitos e 3` `%`
`% ./palitos s 0` `%`	*(termina)*		
	`%`		
		`% ./palitos e 2` *(espera)*	
	`% ./palitos s 1` `%`		
		(termina)	`% ./palitos s 3` `%`
		`%`	
		`% ./palitos s 2` `%`	

Note que el *prompt %* indica cuando un comando termina o si debe esperar. El tiempo avanza hacia abajo. El primer parámetro puede ser *e* para pedir permiso para comenzar a comer o *s* para notificar que se terminó de comer. El segundo parámetro es un entero entre 0 y 4 con el número del filósofo. Recuerde que para poder comer el filósofo *i* necesita los palitos *i* e *(i+1)%5*. En el servidor no necesita programar la función *main*, pero sí debe programar la función que atiende las conexiones de los clientes, declarar todas las variables globales que necesite e indicar su inicialización.

Restricción: Su solución debe evitar la hambruna.

Parte ii.- (1,5 puntos) Programe la siguiente función:

```
int biggest(char *name, int max_len);
```

Esta función debe entregar en *name* el nombre del archivo más grande en el directorio de trabajo (o directorio actual). No considere los subdirectorios. El parámetro *max_len* es el tamaño máximo de *name*. Esta función retorna 0 en caso de éxito o -1 en el caso de que encuentre cualquier error.

El siguiente es un ejemplo de uso de biggest:

```
char buf[256];
if (biggest(buf, 256))
  fprintf(stderr, "error\n");
else
  printf("%s\n", buf);
```