Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ciencias de la Computación e Informática

CI-0117 Programación Paralela y Concurrente
Grupo 01
I Semestre

I Tarea programada: Collatz/Ascensor

Profesor:
Francisco Arroyo

Estudiante:
Jennifer Villalobos Fernández | B67751

15 de mayo del 2020

Índice

1. Introducción

La computación paralela permite el uso simultáneo de múltiples recursos para resolver un problema, proporcionando ciertas ventajas al momento de solucionar problemas que no se podrían realizar en una sola CPU, o que no podrían realizarse en un tiempo razonable. Para poder hacer uso de este modelo de programación se prestan muchas herramientas. En esta tarea utilizaremos dos de ellas (y sus respectivos complementos): creación de múltiples procesos mediante el sistema for() y creación de procesos ligeros o "hilosütilizando la biblioteca pthreads.

Todo programa en ejecución es un proceso y todos los procesos son manejados por el sistema operativo, el cual los diferencia entre sí con un identificador único de proceso, o process id (pid). Cuando nos surge la necesidad de ocupar más de un proceso en un mismo programa, existe la opción de crear otro proceso independiente. Ahí es cuando entra la función fork(), la cual en el momento en que es llamada crea un clon idéntico del proceso que la ejecutó. Sin embargo, a pesar de que ambos procesos son clones, el sistema operativo necesita poder diferenciar entre ellos, y por ende utiliza un pid (una identificación) distinto para cada uno. Para que dos procesos realicen tareas distintas mediante un mismo código, debe utilizarse un condicional (if), así ambos procesos comprobarán si son el proceso hijo o son el proceso padre por medio de la única cosa diferenciable entre ellos: su pid. Teniendo las herramientas para poder realizar múltiples cosas distintas en un mismo código, lo que falta es implementar un segmento de memoria compartida (en esta tarea, por medio de shmget, shmat, shmdt, shmctl...) para poder compartir los elementos en común que se consideren necesarios en nuestro código.

Otra manera de hacer un uso simultáneo de los procesos, es mediante pthreads. Tal y como lo insinúa el nombre, esta biblioteca se basa en el uso de "hilos", los cuales son considerados "procesos ligeros"que representan el segmento de código que está siedo procesado en un momento dado. El uso de hilos en un proceso hace posible ejecutar segmentos de código (que pueden o no ser los mismos) de manera concurrente, permitiendo disminuir el tiempo requerido para realizar una tarea, y sin tener que crear las estructuras de datos que se requieren al momento de la creación de un proceso, ya que los hilos comparten ciertos elementos, como lo son las variables globales. Al igual que el fork() y sus procesos hijos, los hilos también poseen una única identificación, así como sus variables locales y direcciones de retorno.

2. Objetivo:

- Familiarizar al estudiante con el desarrollo de programas concurrentes en ambientes UNIX, al menos con algunos de sus sabores: Linux

3. Descripción

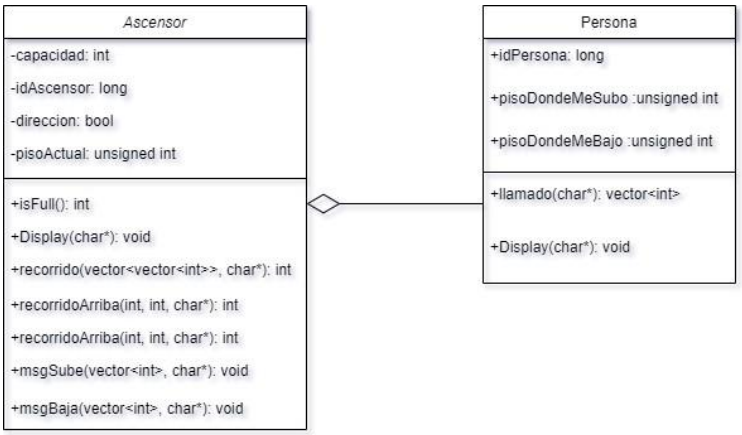
- Collatz: Realizar un programa en C o C++ que reciba un número como parámetro. Suponiendo que el usuario indica el límite superior de un conjunto de valores que quiere calcular, por ejemplo 1000, el conjunto sería entonces [2, 1000]. El programa debe determinar cuál es la mayor cantidad de pasos que Collatz realiza sobre todos los elementos y en cuál elemento del conjunto ocurrieron. Hacer que algunos hijos/hilos estén a cargo de calcular algunos de los términos y luego que el padre obtenga el resultado total de la operación. Para facilidad del usuario el límite siempre será una potencia de 10. El programa debe indicar el tiempo que tarda en realizar las operaciones solicitadas, la cantidad de pasos y el número que los generó.
- Ascensor: Programar en C++ el comportamiento de un ascensor y varias personas utilizando dos tipos de hilos: Persona(e, s) y Ascensor. La sincronización, concurrencia y eficiencia son importantes. Se debe crear al menos una clase para el .Ascensorz otra para las "Personas". Para seguir lo que está ocurriendo en cada momento es menester que tanto el ascensor como las personas emitan mensajes por pantalla de vez en cuando para saber qué es lo que está ocurriendo, por ejemplo:

  - ’Persona 9: Pulso el botón en la planta 3 para subir al piso 10’
  - Persona 4: El ascensor está lleno, no puedo subir’
  - ’Ascensor: Voy bajando por la planta 3’
  - ’Ascensor: Me detengo para que baje un viajero en planta 5’
  - etc...

  Para las pruebas su programa debe funcionar correctamente creando al menos 100 hilos que representen personas. Debe soportar cargas alternas, esto es que podemos generar secuencias de personas (hilos) y permitir que el ascensor comience a trabajar y realizar la atención de personas, luego de un tiempo (sleep x), sin que la primera carga haya terminado arrancar una segunda carga de personas, que en este caso deben acomodarse de acuerdo con la situación actual del ascensor. Al final de la tarea el ascensor debe quedar ocioso y todas las personas atendidas. La generación de las plantas de salida y llegada de las personas debe generarse de manera aleatoria y tener sentido.

4. Diseño

(a) UML de la clase Collatz.



(b) UML del programa Ascensor.

5. Desarrollo

I Parte. Collatz: Para resolver este problema, se utilizó la función fork() y la creación de un segmento de memoria compartida. Para iniciar el programa, se debe digitar el ejecutable y el número límite que el usuario desee ($./nombreEjecutable$ número). Según el enunciado de la tarea, ese numero ingresado debe ser potencia de 10, así que se creó el método Potencia(), el cual se encarga de "filtrar.el número digitado por el usuario y se asegura que este sea potencia de 10 ¿cómo? ya que se busca cúal potencia de 10 es, se le debe calcular el logaritmo en base 10 (función log10). Si el número 10 elevado a la potencia retornada, es igual al numero que ingresó el usuario, entonces este es potencia de 10. Se tomaron estas precauciones ya que con ese dato se puede analizar el número de procesos hijos que pueden llegar a ser necesarios (según la cantidad que ingrese el usuario, entre más grande, más procesos serían necesarios).

Luego de asegurar que el número ingresado es correcto, se analiza cuántos procesos hijos deberían formarse mediante el método Procesos(). Con este dato, también se define la variable rango", la cual divide el número que digitó el usuario entre la cantidad de procesos definida anteriromente, ¿para qué? para que cada proceso hijo haga el cálculo de la conjetura de Collatz en ese rango, así se asegura que todos los procesos trabajan equitativamente y por ende, con un tiempo aproximadamente similar.

Ya definidas las variables necesarias, se inicia la memoria compartida mediante shmget, este segmento posee el tamaño de 2 int (uno, para el elemento con más pasos, el segundo para la cantidad de pasos generados por el mismo).

Se inicia el for para la creación de los procesos hijos. Antes, se toma el id del proceso padre, esto para asegurar que el único creando los procesos hijos, sea él. Al finalizar el for del fork(), cada proceso hijo se une al puntero de memoria compartida mediante shmat.

Ahora lo importante: mediante un condicional, se asegura que los únicos que puedan hacer el cálculo de Collatz sean los procesos hijos. Estos al ingresar se les toma el id ¿cómo? tomando su pid por medio de getpid() y restándole el pid del padre (tomado anteriormente). Esto para que cada proceso hijo quede con su id en términos de 0,1,2,3,... y así se facilite la división del rango. Esta división ocurre de la siguiente manera: el inicio del rango correspondiente a cada hilo se define por la operación: $(rango * son_id) + 1$ y el final de dicho rango por: $rango*(son_id + 1)$ donde son_id representa el id del hijo calculado. Con el rango establecido, cada hijo inicia el for correspondiente para calcular la conjetura de Collatz con todos los elementos que le corresponden. Luego de calcular cada número, un condicional cuestiona si el número calculado posee mayor cantidad de pasos que el dígito que se encuentra en la memoria compartida, si este es el caso, se modifica la memoria, en caso contrario, se deja igual. Este proceso se da con cada hijo.

Cuando todos los hijos terminen su trabajo, estos se separan de la memoria compartida por medio de la función shmdt. El proceso padre espera mientras todos los procesos hijos se eliminan (exit(0)). Cuando solo queda el proceso padre, este se anexa al puntero de memoria compartida mediante shmat, comparte los resultados en consola y elimina la memoria compartida con shmctl. El programa finaliza presentando el tiempo transcurrido durante el programa (calculado con la función clock()).

II Parte. Ascensor: Para iniciar el programa, este se debe ejecutar en terminal con la siguiente línea de comando: ./ejecutable. Para generar este programa se utilizó la biblioteca pthreads. Por medio de esta y utilizando la clase de ejemplo provista por el profesor, se generó un hilo ascensor, ya que según el enunciado, solo había uno en todo el edificio. Además, con la misma lógica para generar el ascensor, se crearon varios hilos que representaban a personas que utilizarían el aparato.

El hilo Persona recibe un identificador como parámetro, el cual se convertirá en su id. Acá se crea una nueva clase Persona, se le asigna el id y se crea un bloque de memoria para compartir en consola los mensajes enviados por esta (por medio de calloc). Este hilo crea un nuevo Mutex, el cual se genera de la clase provista por el profesor, para bloquear y así evitar que alguien más (otro hilo) esté utilizando la pantalla. Este hilo imprime su identificación y el piso donde se encuentra, así como el piso donde desea bajarse. Esto por medio del método Display(). La persona, luego de la introducción, realiza un llamado al ascensor, para que este venga al piso donde se encuentra y poder abordar. Esta tarea se completa por medio del método llamado(), el cual crea un vector (std) de enteros y mediante el comando push back se ingresa su id, el piso de subida y el piso de bajada. Esto nos deja con que cada hilo Persona, es, en realidad, un vector de enteros. Este vector se retorna y se ingresa a la cola de personas solicitando al ascensor. Esta cola es otro vector (esta vez de vectores/personas) declarado en el main como una variable global. Por esta misma razón se utiliza nuevamente el método lock() y unlock() del Mutex, para evitar que otros hilos estén utilizando esta variable compartida.

Ahora entra en juego el hilo Ascensor, este posee un id (que también ingresa como parámetro) y otro bloque de memoria compartida para mostrar sus mensajes. Además, se crea un Mutex y dos int, encargados de tomar los valores retornados por sus métodos. Este hilo posee un ciclo while casi infinito (se detiene hasta que el vector global que almacena todas las solicitudes, esté vacío). Dentro de este ciclo, se inicia el mutex con lock() e inicia su rutina.

Esta rutina consiste en tomar una solicitud de la cola (vector de vectores) y analizar si este hilo puede ingresar o no al ascensor. Si el ascensor está vacío o no se encuentra en el límite de su capacidad (8), este retorna un 0. En caso contrario retorna un 1. En ambos, se imprime en consola la situación dada. Esto se da por medio del método solicitud(). En caso de que el hilo sí pueda ingresar, el hilo ascensor "pasa"(push back) a la persona (el vector) de la solicitud a otro vector de vectores global: sube. Esta estructura almacena todas las personas que pueden ingresar al ascensor, y por ende, hay que ir a buscarlas. Luego de ingresar esa persona al vector de subida, se saca de la cola general de solicitudes.

En este momento el ascensor avisa de su estado: en cúal piso se encuentra y para dónde va. Esto último es un poco más complicado: se realiza por medio del método recorrido(), el cual antes que nada, analiza si el vector sube() está vacío, así como el último vector compartido: baja. Este vector almacena las personas que ya se subieron al ascesor y claramente, necesutan bajar. El método recorrido() luego de analizar los detalles, nos dirá que se debe hacer: se sube a una persona o mejor bajamos a otra, esto con la única condición de ¿qué nos queda más cerca?.

Así que luego de inspeccionar los vectores, si todo está bien y no se encuentran vacíos (en caso de que lo estén retorna un 2, y el hilo ascensor se encarga de admitir 3 nuevas personas para ir por ellas y subirlas), nos fijamos en la dirección del ascensor, si este va hacia arriba, llamamos al método recorridoArriba(), en caso contrario, llámamos a recorridoAbajo(). Estos métodos realizan las mismas funciones, la diferencia es que, si se va para arriba, se debe comparar por cúal piso es mayor que el pisoActual, y si es para abajo, por cúal es menor, y si se tiene que cambiar de dirección, bueno, ambos lo hacen al revés.

Estos métodos de recorrido toman el último elemento de los vectores "suben "bajan". Comparan los pisos respectivos (en el caso de suben, la casilla del vector de interés es 1, en el caso de bajan, es la número 2) y determinan cúal está más cercano, según el pisoActual y según nuestra dirección. En caso de que ambas personas estén en la dirección contraria, el ascensor cambia de dirección y toma al más cercano como objetivo.

Si el más cercano era el hilo de la persona que subía al ascensor, el hilo ascensor ingresa a esta persona (vector) al vector bajan, ya que al estar dentro, ahora necesita bajar, y por ende lo elimina (pop back) del vector suben. En caso contrario (el más cercano era la persona que bajaba), el hilo ascensor elimina a esa persona del vector bajan (pop back). En todo este proceso, el hilo ascensor va notificando de las desiciones tomadas por medio de la consola.

El programa se ejecuta bien hasta cierto momento,no logrando llegar al final, ya que emite un error "Segmentation fault". Por más que busqué, no encontré el error de memoria.

6. Manual de usuario

Requerimientos de Software

- **Sistema Operativo:** Linux.
- **Arquitectura:** 64 bits.
- **Ambiente:** Code::Blocks o Terminal.

Compilación

Para compilar el programa Collatz puede usarse el Makefile, con solo llamar al comando make en terminal. También se puede utilizar `g++` en la siguiente sentencia:

```
$ g++ Collatz.cpp -o collatz
```

Para compilar el programa Ascensor, se utiliza `g++`, así como `lpthread`en la siguiente sentencia:

```
$ g++ Ascensor.cpp Persona.cpp Mutex.cpp main.cpp -lpthread -o nombre_ejecutable
```

Especificación de las funciones del programa

Para ejecutar el programa Collatz, se debe insertar el nombre del ejecutable y la cantidad del límite superior del cálculo, es decir, el número, de esta manera:

```
$ ./collatz 1000
```

Para ejecutar el programa Ascensor, no olvidar insertar el comando `lpthread`.

7. Casos de Prueba

Pruebas Collatz: Estas pruebas verifican el funcionamiento del programa Collatz.
El código utilizado corresponde a:

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <bits/stdc++.h>
#include <math.h>
#include <ctime>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

using namespace std;

int Collatz(int);
bool Potencia(int);
int Procesos(int);

int  main(int  argc, char *argv[])
{
    int limite; //Numero elegido por el usuario
    int cant_pasos; //Cantidad e pasos en que se realizo la conjetura de Collatz.
    int master_id; //El id del proceso padre
    int son_id; //El id del (de los) proceso(s) hijo(s)
    int cant_procs; //Cantidad de procesos por crear
    int rango; //El rango que le corresponde a cada proceso hijo
    int bottom; //Inicio del rango
    int top; //Fin del rango
    clock_t start, finish; //Variables para medir el tiempo.
    double time = 0; //Resultado del tiempo transcurrido

    int    memory_id;
    int    *shared_ptr = {};
    pid_t  pid;
    int    status;

    start = clock();

    //Se le pide al usuario que ingrese un numero.
    if (argc != 2)
    {
        printf("Ingrese: %s <cantidad limite> (que sea potencia de 10)\n", argv[0]);
        exit(1);
    }

    //Si el numero ingresado no es potencia de 10.
    if(!Potencia(atoi(argv[1])))
    {
        printf("El numero ingresado debe ser potencia de 10.\nIngrese: %s <cantidad
            limite> (que sea potencia de 10)\n", argv[0]);
        exit(1);
```

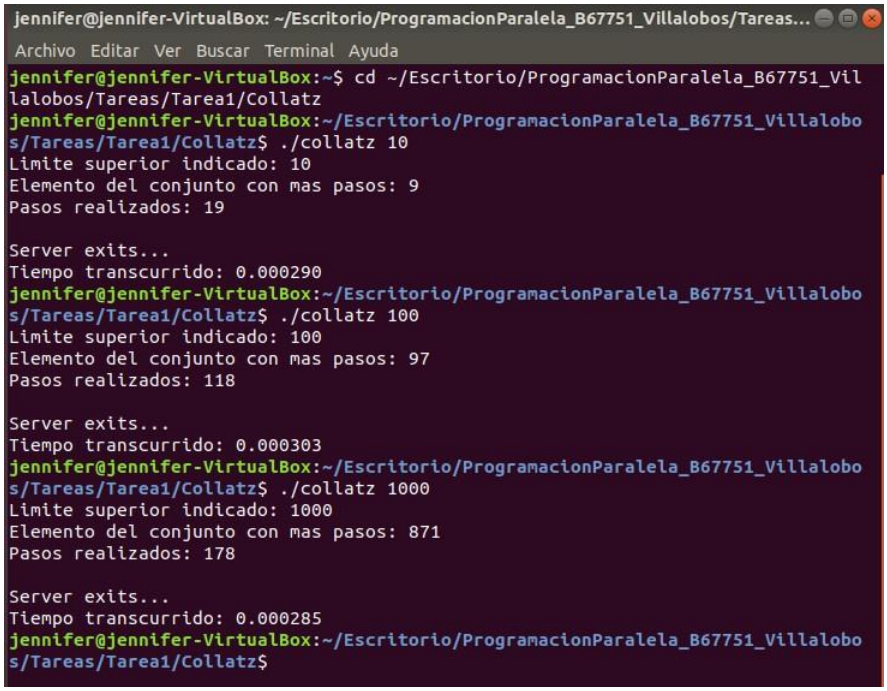
```
    }

    //Se define el rango: 2, limite.
    limite = atoi(argv[1]);
    cant_procs = Procesos(limite);
    rango = limite/cant_procs;
    //Se establece en cero los elementos de la memoria compartida
    //shared_ptr[0] = 0;
    //shared_ptr[1] = 0;


    //Se crea el segmento de memoria compartida
    memory_id = shmget(IPC_PRIVATE, 2*sizeof(int), IPC_CREAT | 0666);
    if (memory_id < 0)
    {
        perror("Error: shmget\n");
        exit(1);
    }


    //Se crean los procesos hijos.
    master_id = getpid(); //Se toma el id del proceso padre

    for(int i=0; i < cant_procs; i++)
    {
        if(getpid() == master_id)
        {
            pid = fork();
            if (pid < 0)
            {
                perror("Error: fork\n");
                exit(1);
            }
        }
        else
        {
            break;
        }
    }

    //Cada hijo se une a la memoria compartida
    shared_ptr = (int *) shmat(memory_id, NULL, 0);
    if ((intptr_t) shared_ptr == -1)
    {
        perror("Error: shmat\n");
        exit(1);
    }


    //Cada hijo hace su parte del calculo.
    if (pid == 0)
    {
        //Se obtiene el id del proceso hijo.
        son_id = (getpid()-master_id)-1;

        //Se define el inicio y el final del rango
```

```
107           //en el que se va a realizar el calculo.
108           //Esto segun el id del proceso.
109           bottom = (rango * son_id) + 1;
110           top = rango * (son_id+1);
111
112           //Si es el proceso 0, el inicio debe tomarse
113           //a partir de 2.
114           if(son_id == 0)
115           {
116
117               for(int i = 2; i <= top; i++)
118               {
119                   cant_pasos = Collatz(i);
120                   if(cant_pasos > shared_ptr[1])
121                   {
122                       shared_ptr[0] = i;
123                       shared_ptr[1] = cant_pasos;
124                   }
125               }
126           }
127           //Si son los demas procesos, el inicio se toma normal.
128           else
129           {
130
131               for(int i = bottom; i <= top; i++)
132               {
133                   cant_pasos = Collatz(i);
134                   if(cant_pasos > shared_ptr[1])
135                   {
136                       shared_ptr[0] = i;
137                       shared_ptr[1] = cant_pasos;
138                   }
139               }
140           }
141       }
142
143
144       //Cada hijo se separa de la memoria compartida.
145       if(shmdt((void *) shared_ptr) == -1)
146       {
147           perror("Error: shmdt\n");
148       }
149
150
151       //Todos los procesos hijo terminan. El padre espera.
152       if(getpid() == master_id)
153       {
154           for(int i=0; i < cant_procs; i++)
155           {
156               wait(&status);
157           }
158       }
159       else
160       {
161           exit(0);
162       }
```

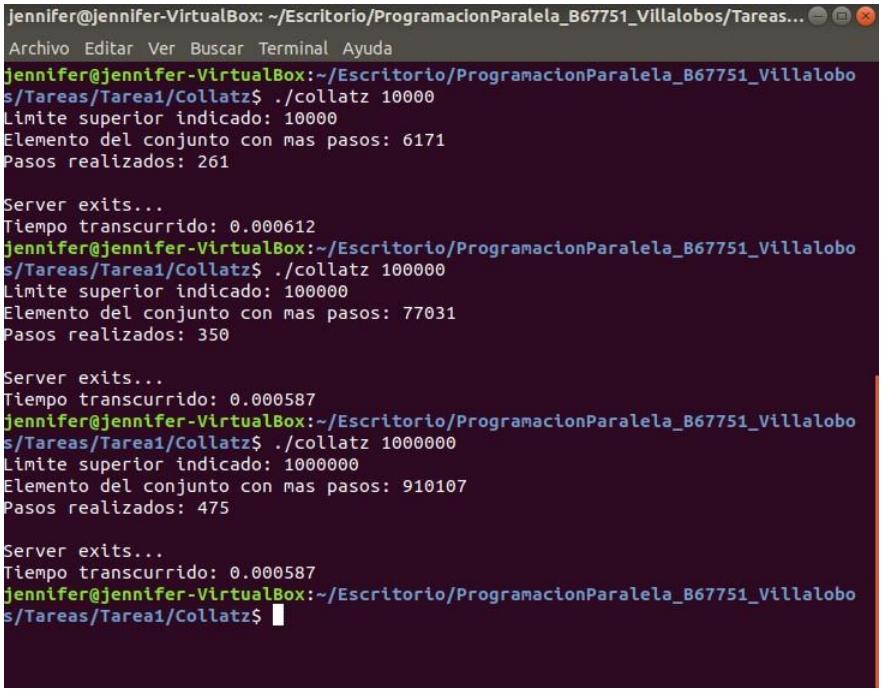
```
163
164         //Se anexa el puntero al proceso master.
165         shared_ptr = (int*) shmat(memory_id, NULL, 0);
166         //Se muestra el resultado
167         printf("Limite superior indicado: %d \nElemento del conjunto con mas pasos: %d \
                nPasos realizados: %d \n\n",
168                 limite, shared_ptr[0], shared_ptr[1]);
169
170         //El proceso master elimina la memoria compartida.
171         shmctl(memory_id, IPC_RMID, NULL);
172
173         //Se calcula el tiempo de ejecucion
174         finish = clock();
175         time = finish-start;
176         time = double(time/CLOCKS_PER_SEC);
177
178         printf("Server exits...\nTiempo transcurrido: %f\n", time);
179         exit(0);
180     }
181
182     /*
183     *Metodo encargado de realizar la conjetura de
184     *Collatz con el termino que ingresa como parametro.
185     */
186     int Collatz(int rango)
187     {
188         int cant_pasos = 0;
189         int num_actual = rango;
190
191         while(num_actual > 1)
192         {
193             //Si el numero es impar
194             if((num_actual%2) != 0)
195             {
196                 num_actual = (num_actual*3)+1;
197             }
198             //Si el numero es par
199             else
200             {
201                 num_actual = num_actual/2;
202             }
203             cant_pasos++;
204         }
205
206         //Retorna la cantidad de pasos llevados a cabo para
207         //llegar al numero 1.
208         return cant_pasos;
209     }
210
211     /*
212     *Metodo encargado de calcular los hijos/procesos
213     *necesarios, segun el limite superior ingresado
214     *por el usuario.
215     */
216     int Procesos(int limite)
217     {
```

```
    int cant_procs;
    int potencia;

//Se obtiene la potencia de 10 correspondiente
//del numero ingresado.
    potencia = log10(limite);

    //Si el numero es 10, 100 o 1000: 2 hijos.
    if(potencia <= 3)
    {
        cant_procs = 2;
    }
    //Si el numero va desde 10000 a un millon: 5 hijos.
    else if(potencia > 3 && potencia < 7)
    {
        cant_procs = 5;
    }
    //Si el numero es 10 millones o 100 millones: 10 hijos.
    else if(potencia > 7 && potencia < 9)
    {
        cant_procs = 10;
    }
    //Si el numero es mayor: 20 hijos.
    else
    {
        cant_procs = 20;
    }

    return cant_procs;
}


/*
*Metodo que revisa si el numero ingresado es una potencia de 10.
*/
bool Potencia(int numero)
{
    int potencia = log10(numero);

    //Si el resultado elevado a la potencia
    //es igual al numero entrante, este es
    //potencia de 10.
    if(pow(10, potencia) == numero)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

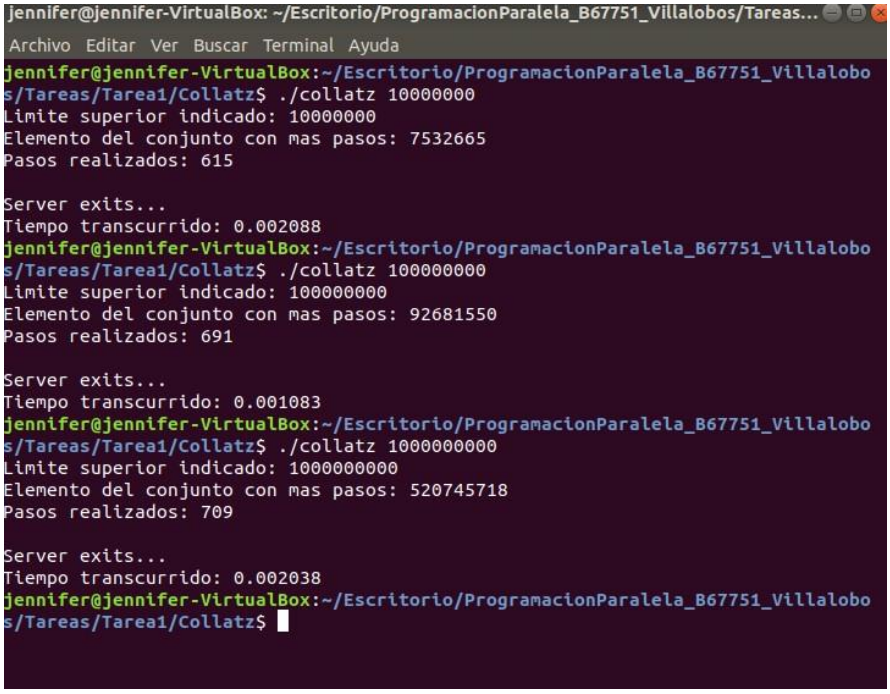
Los resultados obtenidos con las cifras solicitadas ($10, 100, 1000, 10000, 10^5, 10^6, 10^7, 10^8, 10^9$) se pueden verificar en las siguientes figuras:



(c) Salida del programa con las cantidades 10, 100 y 1000.

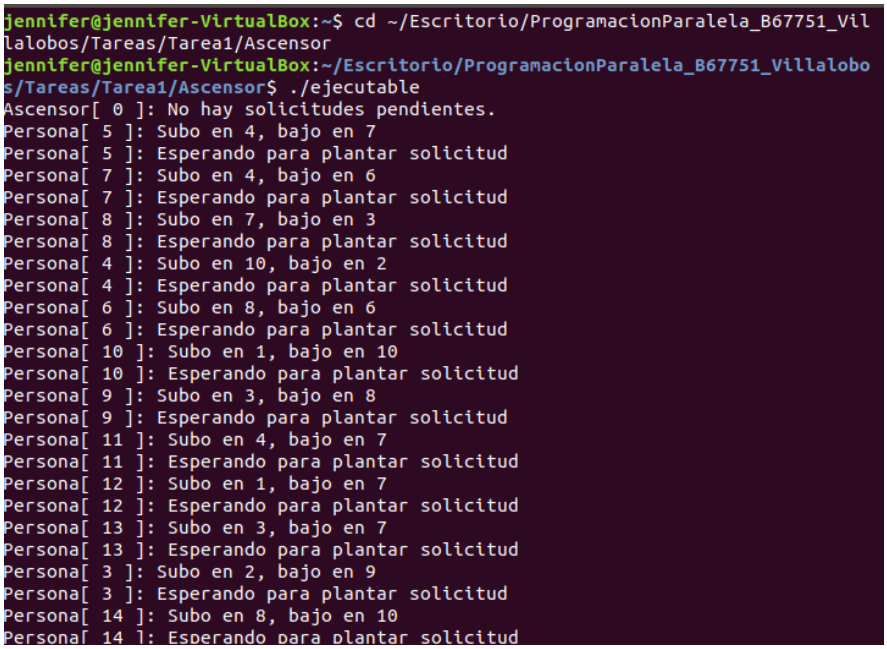


(d) Salida del programa con las cantidades diez mil, cien mil y un millón.

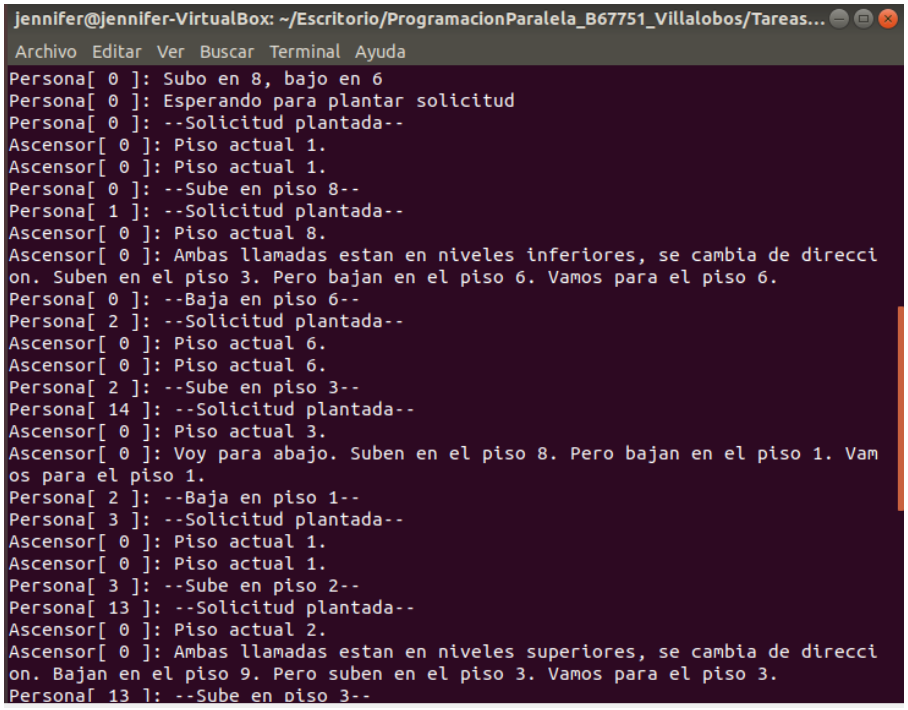


(e) Salida del programa con las cantidades diez millones, cien millones y un billón.

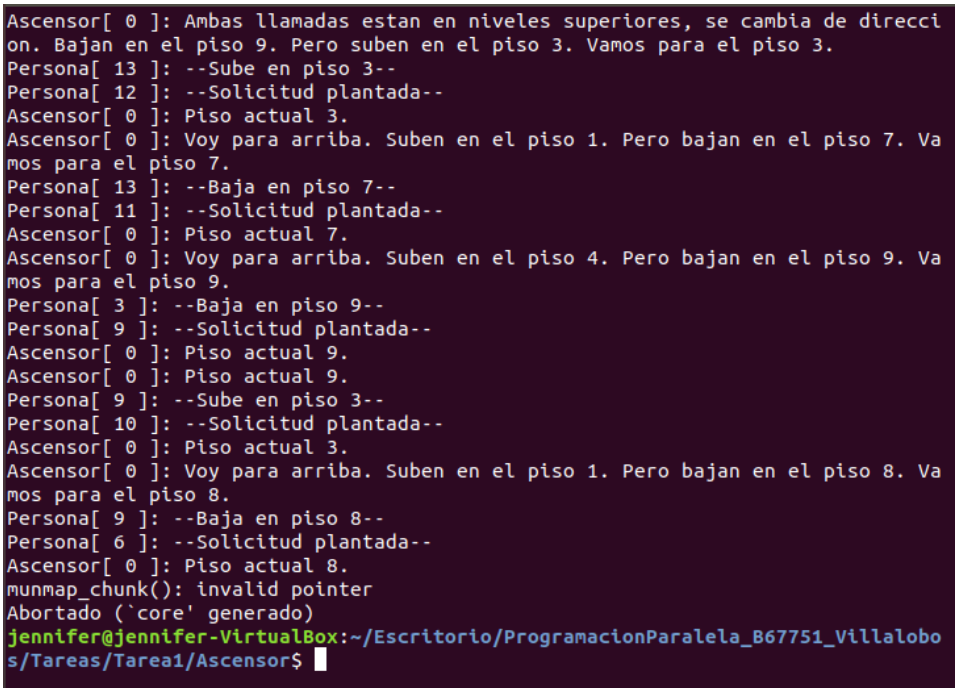
Estas capturas verifican el funcionamiento del programa Ascensor.



(f) Salida del programa con 15 personas solicitantes y 1 ascensor.



(g) Intervención del ascensor.



(h) Error obtenido al tiempo de ejecutar el programa.