ecet4640-lab4

Generated by Doxygen 1.9.3

Chapter 1

ecet4640-lab4

This program reads user information using the `who` command and publishes that information to shared virtual memory for client processes to read. It updates every second.

The main.c page is a good starting point for following the call graph.

Here is a general call graph at a glance:

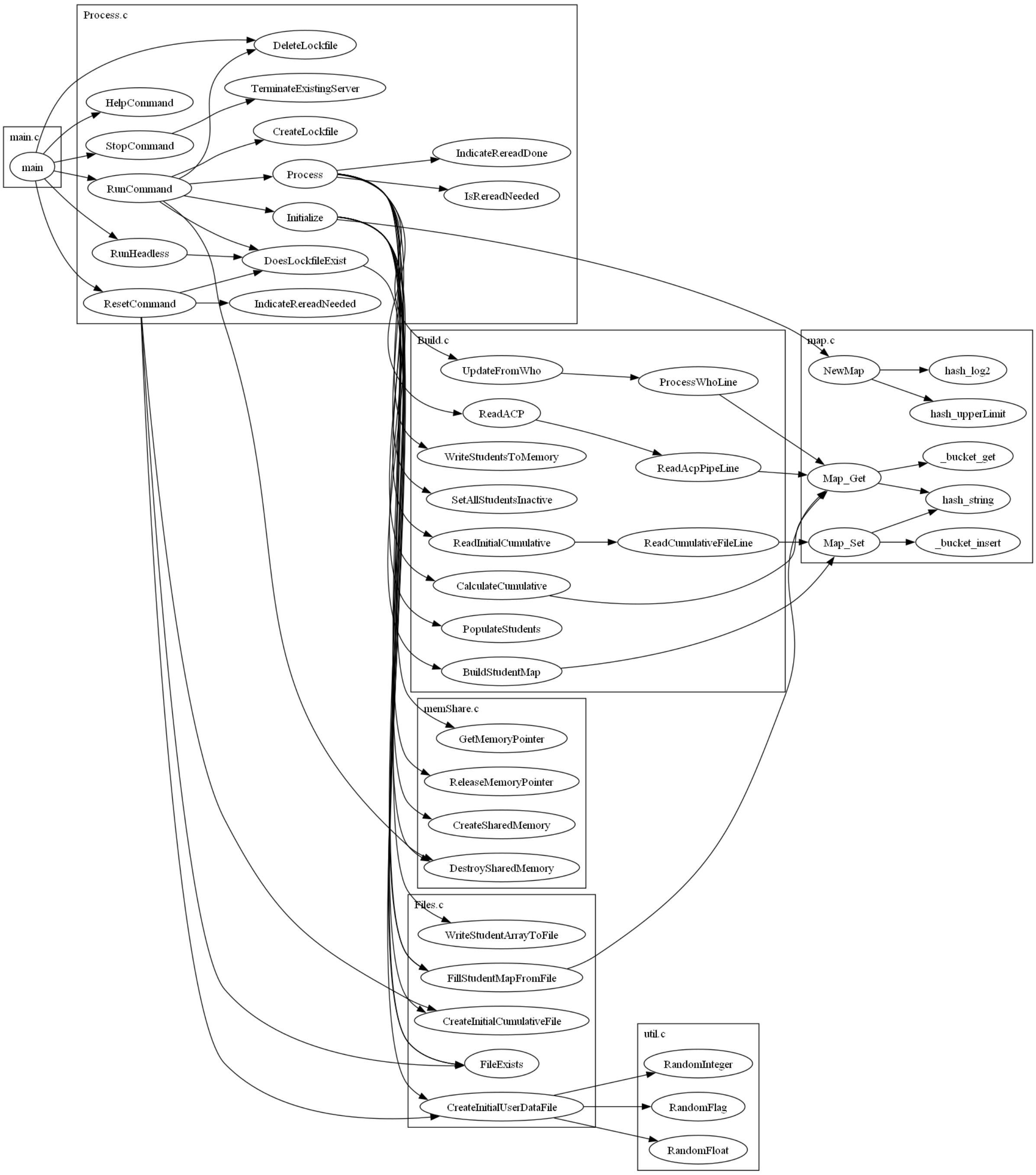


Figure 1.1 Program Call graph

1.0.1 About

Created for Computer Networking, September 2023, by `Karl Miller`, Paul Shriner, and Christian Messmer.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

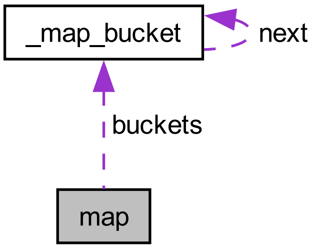
Data Structure Documentation

4.1 map Struct Reference

A map. Stores key-value pairs for near constant lookup and insertion time.

`#include <map.h>`

Collaboration diagram for map:



Data Fields

- int size
- struct _map_bucket ∗ buckets

4.1.1 Detailed Description

A map. Stores key-value pairs for near constant lookup and insertion time.

Note

Use `NewMap` to create a new map.

Use Map_Set to set a key in the map.

Use Map_Get to get a value from the map.

The values stored are of type void pointer.

Definition at line 99 of file map.h.

4.1.2 Field Documentation

4.1.2.1 buckets

```
struct _map_bucket* buckets
```

Definition at line 104 of file map.h.

4.1.2.2 size

```
int size
```

Definition at line 102 of file map.h.

The documentation for this struct was generated from the following file:

- src/server/map.h

4.2 map_result Struct Reference

The result of a map retrieval.

```
#include <map.h>
```

Data Fields

- short found
- void * data

4.2.1 Detailed Description

The result of a map retrieval.

Definition at line 110 of file map.h.

4.2.2 Field Documentation

4.2.2.1 data

```
void* data
```

Definition at line 115 of file map.h.

4.2.2.2 found

```
short found
```

Definition at line 113 of file map.h.

The documentation for this struct was generated from the following file:

- src/server/map.h

4.3 Student Struct Reference

The student data type.

```
#include <Data.h>
```

Data Fields

- char userID [DATA_ID_MAX_LENGTH]
- char fullName [DATA_NAME_MAX_LENGTH]
- short age
- float gpa
- short active
- time_t lastLogin
- int loginDuration

4.3.1 Detailed Description

The student data type.

Definition at line 38 of file Data.h.

4.3.2 Field Documentation

4.3.2.1 active

```
short active
```

Definition at line 44 of file Data.h.

4.3.2.2 age

```
short age
```

Definition at line 42 of file Data.h.

4.3.2.3 fullName

```
char fullName[DATA_NAME_MAX_LENGTH]
```

Definition at line 41 of file Data.h.

4.3.2.4 gpa

```
float gpa
```

Definition at line 43 of file Data.h.

4.3.2.5 lastLogin

```
time_t lastLogin
```

Definition at line 45 of file Data.h.

4.3.2.6 loginDuration

```
int loginDuration
```

Definition at line 46 of file Data.h.

4.3.2.7 userID

```
char userID[DATA_ID_MAX_LENGTH]
```

Definition at line 40 of file Data.h.

The documentation for this struct was generated from the following file:

- src/server/Data.h

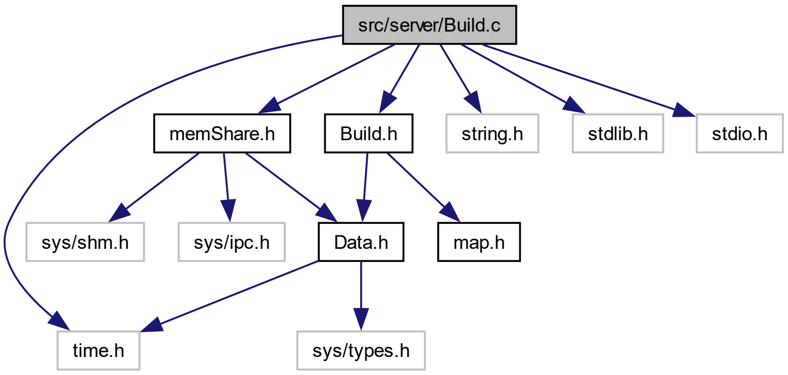
Chapter 5

File Documentation

5.1 src/server/Build.c File Reference

```
#include "Build.h"
#include "memShare.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
```

Include dependency graph for Build.c:



Functions

- void PopulateStudents (char ∗∗studentIDs, char ∗∗studentNames, int arsize)
- void BuildStudentMap (map ∗stmap, Student ∗studentArr, int studentArrLength)
- int UpdateFromWho (map ∗stmap)
- int ProcessWhoLine (map ∗stmap, char ∗whoLine, int whoLineLength)
- void SetAllStudentsInactive (Student ∗stud_arr, int arr_len)
- void WriteStudentsToMemory (void ∗mem_ptr, Student ∗stud_arr, int arr_len)
- int ReadInitialCumulative (map ∗time_map, char ∗filename)
- int ReadACP (map ∗st_map)
- void ReadCumulativeFileLine (map ∗cum_map, char ∗acp_line)
- int ReadAcpPipeLine (map ∗stmap, char ∗acp_line)
- void CalculateCumulative (Student ∗stud_arr, int stud_arr_len, map ∗cum_map)

Variables

- Student ∗ students

  Definitions for functions that populate data structures.
- short dirty = 1

5.1.1 Function Documentation

5.1.1.1 BuildStudentMap()

```
void BuildStudentMap (
            map ∗ stmap,
            Student ∗ studentArr,
            int studentArrLength )
```

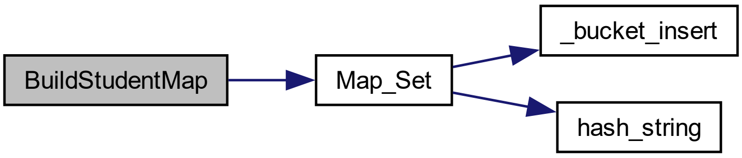
Given a student array, populates a student map, where the student IDs are the key, and the values are pointers to the items in the array.

Parameters

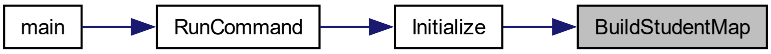
map	The map structure to populate.
studentArr	An array of student structures.
studentArrLength	The length of the students array.

Definition at line 27 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.1.2 CalculateCumulative()

```
void CalculateCumulative (
            Student * stud_arr,
            int stud_arr_len,
            map * cum_map )
```

Calculates the cumulative time for each student by subtracting map[studentID] from student.loginDuration.

Warning

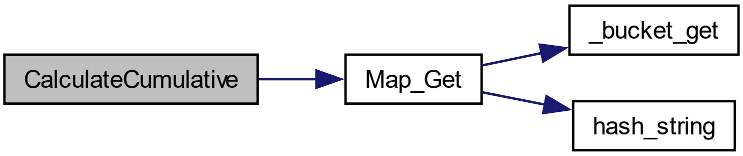
student.loginDuration must have already been set to the total cumulative time logged in.

Parameters

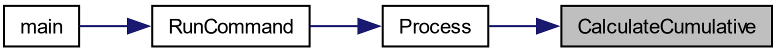
stud_arr	The student's array.
arr_len	The length of students array.
cum_map	A map mapping studentIds to their cumulative login time when the server was started.

Definition at line 205 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.1.3 PopulateStudents()

```
void PopulateStudents (
             char ** studentIDs,
             char ** studentNames,
             int arsize )
```

Allocate and populate the Students array with data.

Parameters

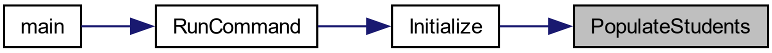
studentIDs	An array of student IDs.
studentNames	An array of student names.
arsize	The size of the array to allocate.

Warning

> studentIDs and studentNames must both be arsize in length.

Definition at line 15 of file Build.c.

Here is the caller graph for this function:



5.1.1.4 ProcessWhoLine()

```
int ProcessWhoLine (
             map * stmap,
             char * whoLine,
             int whoLineLength )
```

Processes a single line as read from the 'who' shell command. Uses that data to update the relevant student by retrieving them from the student map. Updates that students last login time. Also sets 'active' to 1 for the found student.

Attention

> May set dirty to 1.

Parameters

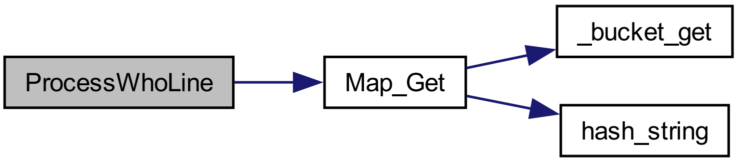
stmap	The student map.
whoLine	The line of text, such as returned from fgets
whoLineLength	The length of that text.

Returns

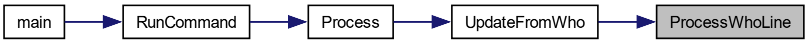
0 if success, -1 if the student was not found in the map.

Definition at line 60 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.1.5 ReadACP()

```
int ReadACP (
            map * st_map )
```

Pipes ac -p, then calls ReadCumulativeLine to update the student map.

Note

After this runs, the student map cumulative will be their total login time in the system. This total time must be subtracted from the cumulative map time to find the time they have been logged in since the program started.

Parameters

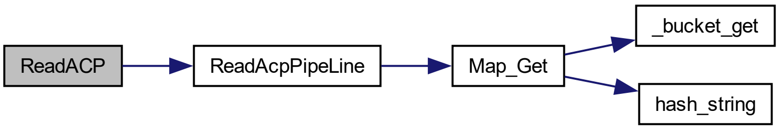
st_map	The students map.

Returns

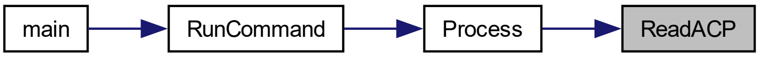
0 on success.

Definition at line 152 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.1.6 ReadAcpPipeLine()

```
int ReadAcpPipeLine (
             map * stmap,
             char * acp_line )
```

Reads a single line from the result of ac -p into the students map.

Parameters

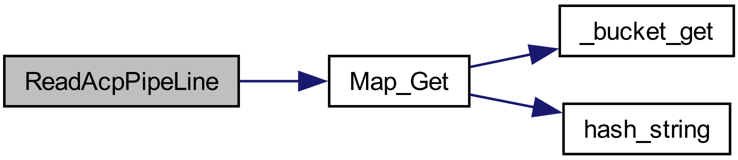
stmap	A map of students.
acp_line	A string representing 1 line result from ac -p.

Returns

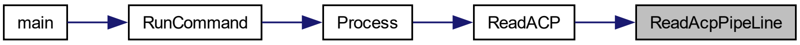
-1 if acp_line is NULL or length is less than 1, otherwise 0.

Definition at line 186 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.1.7 ReadCumulativeFileLine()

```
void ReadCumulativeFileLine (
            map * cum_map,
            char * acp_line )
```

Reads a single line from the initial cumulative file and updates the map so that userID maps to a float value in the initial file.

Note

A line is structured like this: `mes08346 10.06` It finishes with a line starting with `total`; this line should be disregarded.

Parameters

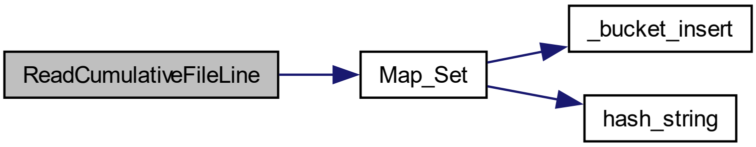
cum_map	The cumulative map.
acp_line	A single line from ac -p.

Returns

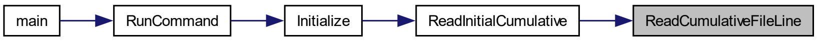
-1 ...

Definition at line 176 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.1.8 ReadInitialCumulative()

```
int ReadInitialCumulative (
            map * time_map,
            char * filename )
```

Populates the cumulative map by reading from the initial cumulative file. The map will be of the form [userID] -> minutes_float

The map will contain users who we don't care about, but it doesn't matter.

Parameters

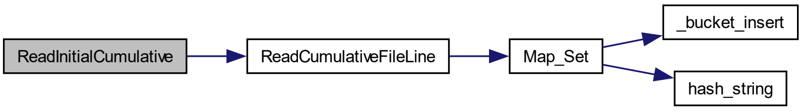
cum_map	A map of cumulative times. Different from the students map.
filename	The filename where the initial cumulative times are located.

Returns

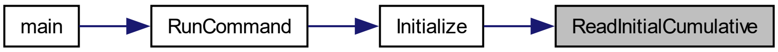
0 if success. -1 if it failed to find the file.

Definition at line 135 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.1.9 SetAllStudentsInactive()

```
void SetAllStudentsInactive (
            Student * stud_arr,
            int arr_len )
```

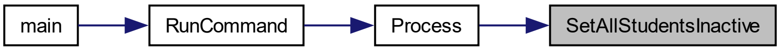
Sets the 'active' member on all students in the students array to 0.

Parameters

stud_arr	The students array.
arr_len	The length of the students array.

Definition at line 108 of file Build.c.

Here is the caller graph for this function:



5.1.1.10 UpdateFromWho()

```
int UpdateFromWho (
             map * stmap )
```

Executes the 'who' command by reading from a file pipe. Calls ProcessWhoLine for each line, to realize updates in the user data from the who command.

Parameters

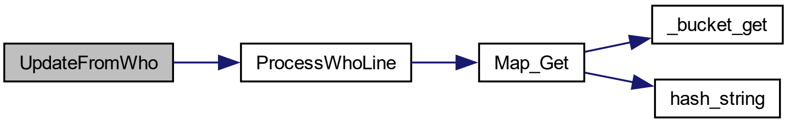
stmap	The student map.

Returns

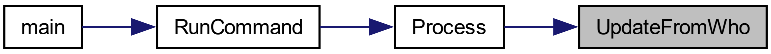
0 if succesful, otherwise nonzero.

Definition at line 40 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.1.1.11 WriteStudentsToMemory()

```
void WriteStudentsToMemory (
             void * mem_ptr,
             Student * stud_arr,
             int arr_len )
```

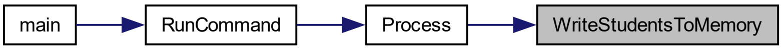
Writes the students array to the location specified by mem_ptr (eg. the shared memory segment).

Parameters

mem_ptr	The address to write at.
stud_arr	The students array to write.
arr_len	The length of the students array.

Definition at line 117 of file Build.c.

Here is the caller graph for this function:



5.1.2 Variable Documentation

5.1.2.1 dirty

```
short dirty = 1
```

Set to '1' if there are changes that should be written to a file.

Definition at line 38 of file Build.c.

5.1.2.2 students

```
Student* students
```

Definitions for functions that populate data structures.

Declarations for functions that populate data structures.

Definition at line 13 of file Build.c.

5.2 Build.c

Go to the documentation of this file.

```

#include "Build.h"
#include "memShare.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// ~~~~~~~~  Data Structures ~~~~~~~~~

Student *students;

void PopulateStudents(char **studentIDs, char **studentNames, int arsize)
{
    students = malloc(sizeof(Student) * arsize);
    int i;
    for (i = 0; i < arsize; i++)
    {
        strcpy(students[i].userID, studentIDs[i]);
        strcpy(students[i].fullName, studentNames[i]);
        // students[i].age = randAge(18, 22);
    }
}

void BuildStudentMap(map *stmap, Student *studentArr, int studentArrLength)
{
    int i;
    for (i = 0; i < studentArrLength; i++)
    {
        Map_Set(stmap, studentArr[i].userID, (void *)(&studentArr[i]));
    }
}

// ~~~~~~~~  Processing ~~~~~~~~~

short dirty = 1; // start dirty

int UpdateFromWho(map *stmap)
{
    char command[4] = "who";
    char line[100];
    FILE *fpipe;
    fpipe = popen(command, "r");
    if (fpipe == NULL)
    {
        return -1;
    }

    while (fgets(line, sizeof(line), fpipe) != NULL)
    {
        ProcessWhoLine(stmap, line, strlen(line));
    }
    pclose(fpipe);

    return 0;
}

int ProcessWhoLine(map *stmap, char *whoLine, int whoLineLength)
{
    char userId[20];
    char dateString[50];
    char timeString[20];
    int read_total = 0;
    int read;
    sscanf(whoLine, " %s %n", userId, &read);
    read_total += read;

    map_result mr = Map_Get(stmap, userId);
    if (!mr.found)
    { // if we can’t find that person in the map, return early
        return -1;
    }
    Student *student = (Student *)mr.data;

    sscanf(whoLine + read_total, " %s %n", dateString, &read); // will be thrown away.  eg ‘pts/1’
    read_total += read;
    sscanf(whoLine + read_total, " %s %n", dateString, &read); // read the date string
    read_total += read;
    sscanf(whoLine + read_total, " %s %n", timeString, &read); // read the time string
    strcat(dateString, " ");
    strcat(dateString, timeString); // catenate the time string back to the date string

```

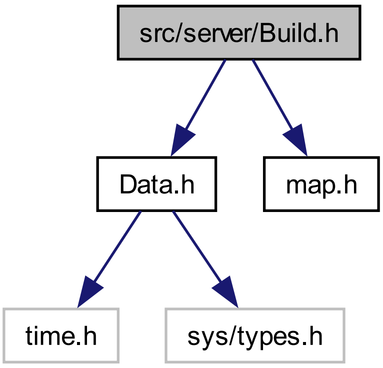
```c
    time_t now = time(NULL);
    struct tm dtime = *localtime(&now);
    dtime.tm_sec = 0;

    memset(&dtime, 0, sizeof(struct tm));

    sscanf(dateString, "%d-%d-%d %d:%d", &(dtime.tm_year), &(dtime.tm_mon), &(dtime.tm_mday), &(dtime.tm_hour), &(dtime.tm_min));

    dtime.tm_year -= 1900;
    dtime.tm_mon -= 1;
    dtime.tm_hour -= 1;

    time_t parsed_time = mktime(&dtime);

    if (student->lastLogin != parsed_time)
    {
        student->lastLogin = parsed_time;
        dirty = 1;
    }
    student->active = 1;
    return 0;
}

void SetAllStudentsInactive(Student *stud_arr, int arr_len)
{
    int i;
    for (i = 0; i < arr_len; i++)
    {
        stud_arr[i].active = 0;
    }
}

void WriteStudentsToMemory(void *mem_ptr, Student *stud_arr, int arr_len)
{
    Student *memloc = (Student *)mem_ptr;
    int i;
    for (i = 0; i < arr_len; i++)
    {
        strcpy(memloc[i].userID, stud_arr[i].userID);
        strcpy(memloc[i].fullName, stud_arr[i].fullName);
        memloc[i].age = stud_arr[i].age;
        memloc[i].gpa = stud_arr[i].gpa;
        memloc[i].active = stud_arr[i].active;
        memloc[i].lastLogin = stud_arr[i].lastLogin;
        memloc[i].loginDuration = stud_arr[i].loginDuration;
    }
}

// ~~~~~~~~  Cumulative Processing ~~~~~~~~~

int ReadInitialCumulative(map *time_map, char *filename)
{
    FILE *file = fopen(filename, "r");
    char line[100];
    if (file == NULL)
    {
        return -1;
    }
    while (fgets(line, sizeof(line), file) != NULL)
    {
        ReadCumulativeFileLine(time_map, line);
    }

    fclose(file);
    return 0;
}

int ReadACP(map *st_map)
{
    char command[6] = "ac -p";
    char line[300];
    FILE *fpipe;
    fpipe = popen(command, "r");
    if (fpipe == NULL)
    {
        return -1;
    }
    int err;
    while (fgets(line, sizeof(line), fpipe) != NULL)
    {
        err = ReadAcpPipeLine(st_map, line);
        if (err)
        {
            printf("\nError %d reading acp pipeline.", err);
            break;
        }
```

```
    }
    pclose(fpipe);
    return 0;
}

void ReadCumulativeFileLine(map *cum_map, char *acp_line)
{
    char userId[20];
    float minutes;
    sscanf(acp_line, " %s %f ", userId, &minutes);
    // int seconds = (int) (minutes * 60)
    long seconds = (long)(minutes * 60);
    Map_Set(cum_map, userId, (void *)seconds);
}

int ReadAcpPipeLine(map *stmap, char *acp_line)
{
    if (acp_line == NULL || strlen(acp_line) < 1)
    {
        return -1;
    }
    char userId[40];
    float minutes;
    sscanf(acp_line, "%s %f", userId, &minutes);
    map_result result = Map_Get(stmap, userId);
    if (result.found)
    {
        Student *student = (Student *)result.data;
        int seconds = (int)(minutes * 60);
        student->loginDuration = seconds;
    }
    return 0;
}

void CalculateCumulative(Student *stud_arr, int stud_arr_len, map *cum_map)
{
    int i;
    for (i = 0; i < stud_arr_len; i++)
    {
        map_result result = Map_Get(cum_map, stud_arr[i].userID);
        if (result.found)
        {
            long time_at_server_start = (long)result.data;
            stud_arr[i].loginDuration = stud_arr[i].loginDuration - time_at_server_start;
        }
    }
}
```

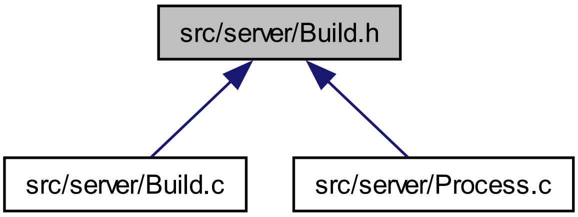
5.3 src/server/Build.h File Reference

```
#include "Data.h"
#include "map.h"
```

Include dependency graph for Build.h:



This graph shows which files directly or indirectly include this file:



Functions

- void PopulateStudents (char ∗∗studentIDs, char ∗∗studentNames, int arsize)
- void BuildStudentMap (map ∗stmap, Student ∗studentArr, int studentArrLength)
- int UpdateFromWho (map ∗stmap)
- int ProcessWhoLine (map ∗stmap, char ∗whoLine, int whoLineLength)
- void SetAllStudentsInactive (Student ∗stud_arr, int arr_len)
- void WriteStudentsToMemory (void ∗mem_ptr, Student ∗stud_arr, int arr_len)
- int ReadInitialCumulative (map ∗time_map, char ∗filename)
- int ReadACP (map ∗st_map)
- void ReadCumulativeFileLine (map ∗cum_map, char ∗acp_line)
- int ReadAcpPipeLine (map ∗stmap, char ∗acp_line)
- void CalculateCumulative (Student ∗stud_arr, int stud_arr_len, map ∗cum_map)

Variables

- Student ∗ students

    Declarations for functions that populate data structures.
- short dirty

5.3.1 Function Documentation

5.3.1.1 BuildStudentMap()

```
void BuildStudentMap (
            map ∗ stmap,
            Student ∗ studentArr,
            int studentArrLength )
```

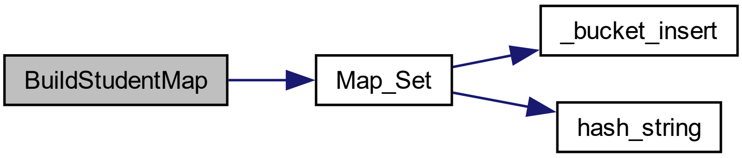
Given a student array, populates a student map, where the student IDs are the key, and the values are pointers to the items in the array.

Parameters

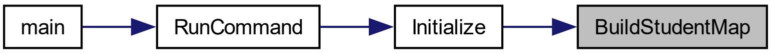
map	The map structure to populate.
studentArr	An array of student structures.
studentArrLength	The length of the students array.

Definition at line 27 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.3.1.2 CalculateCumulative()

```
void CalculateCumulative (
            Student * stud_arr,
            int stud_arr_len,
            map * cum_map )
```

Calculates the cumulative time for each student by subtracting map[studentID] from student.loginDuration.

Warning

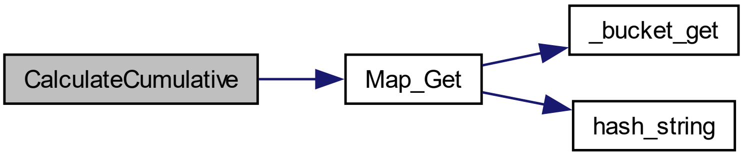
student.loginDuration must have already been set to the total cumulative time logged in.

Parameters

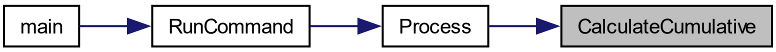
stud_arr	The student's array.
arr_len	The length of students array.
cum_map	A map mapping studentIds to their cumulative login time when the server was started.

Definition at line 205 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.3.1.3 PopulateStudents()

```
void PopulateStudents (
             char ** studentIDs,
             char ** studentNames,
             int arsize )
```

Allocate and populate the Students array with data.

Parameters

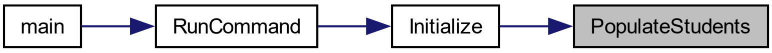
studentIDs	An array of student IDs.
studentNames	An array of student names.
arsize	The size of the array to allocate.

Warning

studentIDs and studentNames must both be arsize in length.

Definition at line 15 of file Build.c.

Here is the caller graph for this function:



5.3.1.4 ProcessWhoLine()

```
int ProcessWhoLine (
             map * stmap,
             char * whoLine,
             int whoLineLength )
```

Processes a single line as read from the 'who' shell command. Uses that data to update the relevant student by retrieving them from the student map. Updates that students last login time. Also sets 'active' to 1 for the found student.

Attention

May set dirty to 1.

Parameters

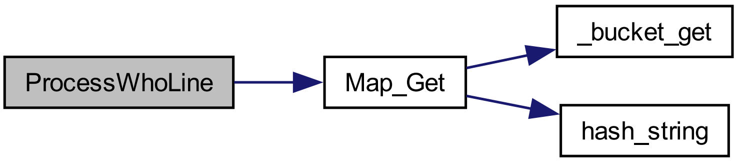
stmap	The student map.
whoLine	The line of text, such as returned from fgets
whoLineLength	The length of that text.

Returns

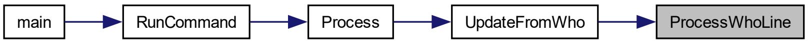
0 if success, -1 if the student was not found in the map.

Definition at line 60 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.3.1.5 ReadACP()

```
int ReadACP (
            map * st_map )
```

Pipes ac -p, then calls ReadCumulativeLine to update the student map.

Note

After this runs, the student map cumulative will be their total login time in the system. This total time must be subtracted from the cumulative map time to find the time they have been logged in since the program started.

Parameters

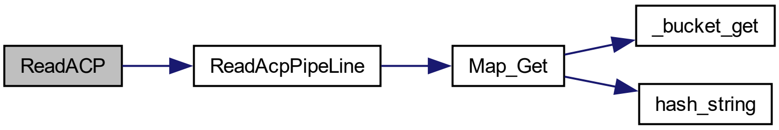
st_map	The students map.

Returns

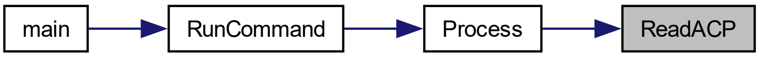
0 on success.

Definition at line 152 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.3.1.6 ReadAcpPipeLine()

```
int ReadAcpPipeLine (
             map * stmap,
             char * acp_line )
```

Reads a single line from the result of ac -p into the students map.

Parameters

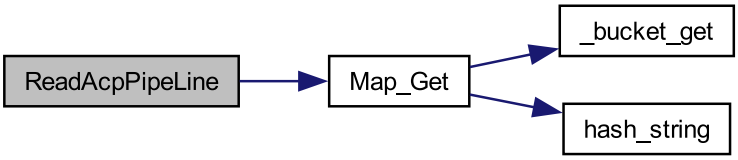
stmap	A map of students.
acp_line	A string representing 1 line result from ac -p.

Returns

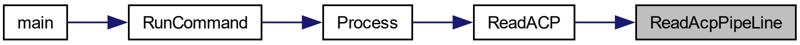
-1 if acp_line is NULL or length is less than 1, otherwise 0.

Definition at line 186 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.3.1.7 ReadCumulativeFileLine()

```
void ReadCumulativeFileLine (
            map * cum_map,
            char * acp_line )
```

Reads a single line from the initial cumulative file and updates the map so that userID maps to a float value in the initial file.

Note

A line is structured like this: `mes08346 10.06` It finishes with a line starting with `total`; this line should be disregarded.

Parameters

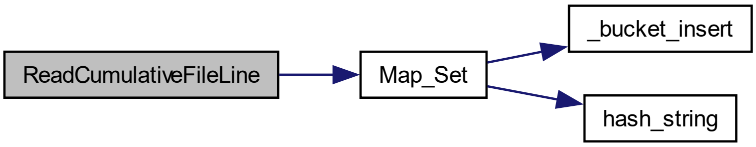
cum_map	The cumulative map.
acp_line	A single line from ac -p.

Returns

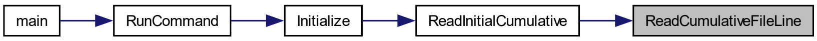
-1 ...

Definition at line 176 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.3.1.8 ReadInitialCumulative()

```
int ReadInitialCumulative (
            map * time_map,
            char * filename )
```

Populates the cumulative map by reading from the initial cumulative file. The map will be of the form [userID] -> minutes_float

The map will contain users who we don't care about, but it doesn't matter.

Parameters

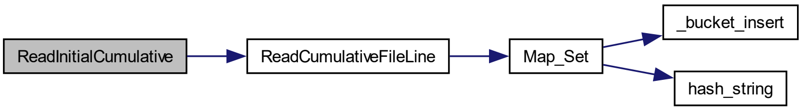
cum_map	A map of cumulative times. Different from the students map.
filename	The filename where the initial cumulative times are located.

Returns

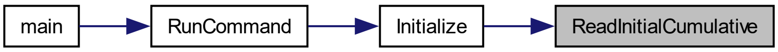
0 if success. -1 if it failed to find the file.

Definition at line 135 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.3.1.9 SetAllStudentsInactive()

```
void SetAllStudentsInactive (
             Student * stud_arr,
             int arr_len )
```

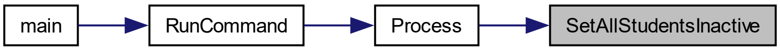
Sets the 'active' member on all students in the students array to 0.

Parameters

stud_arr	The students array.
arr_len	The length of the students array.

Definition at line 108 of file Build.c.

Here is the caller graph for this function:



5.3.1.10 UpdateFromWho()

```
int UpdateFromWho (
            map * stmap )
```

Executes the 'who' command by reading from a file pipe. Calls ProcessWhoLine for each line, to realize updates in the user data from the who command.

Parameters

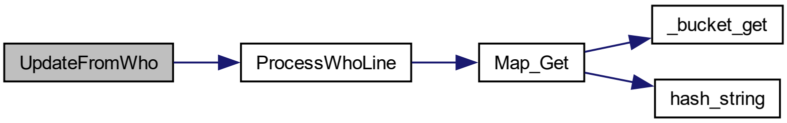
stmap	The student map.

Returns

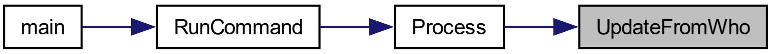
0 if succesful, otherwise nonzero.

Definition at line 40 of file Build.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.3.1.11 WriteStudentsToMemory()

```
void WriteStudentsToMemory (
            void * mem_ptr,
            Student * stud_arr,
            int arr_len )
```

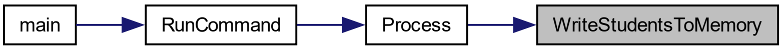
Writes the students array to the location specified by mem_ptr (eg. the shared memory segment).

Parameters

mem_ptr	The address to write at.
stud_arr	The students array to write.
arr_len	The length of the students array.

Definition at line 117 of file Build.c.

Here is the caller graph for this function:



5.3.2 Variable Documentation

5.3.2.1 dirty

```
short dirty [extern]
```

Set to '1' if there are changes that should be written to a file.

Definition at line 38 of file Build.c.

5.3.2.2 students

```
Student* students [extern]
```

Declarations for functions that populate data structures.

The underlying students array. Will be heap allocated with malloc, after PopulateStudents is called.

Generally this array and its length are still passed around via parameters, to decouple as much as possible and enable simple testing and dummy data.

Declarations for functions that populate data structures.

Definition at line 13 of file Build.c.

5.4 Build.h

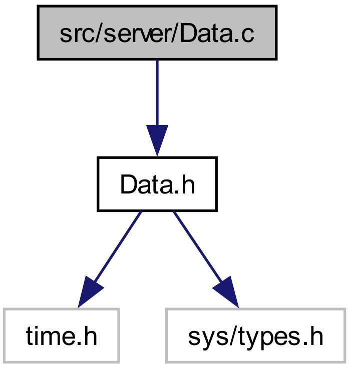
Go to the documentation of this file.

```c
#ifndef BUILD_H
#define BUILD_H
#include "Data.h"
#include "map.h"

// ~~~~~~~~  Data Structures ~~~~~~~~~

extern Student *students;

void PopulateStudents(char **studentIDs, char **studentNames, int arsize);

void BuildStudentMap(map *stmap, Student *studentArr, int studentArrLength);

// ~~~~~~~~  Processing ~~~~~~~~~

extern short dirty;

int UpdateFromWho(map *stmap);

int ProcessWhoLine(map *stmap, char *whoLine, int whoLineLength);

void SetAllStudentsInactive(Student *stud_arr, int arr_len);

void WriteStudentsToMemory(void *mem_ptr, Student *stud_arr, int arr_len);

// ~~~~~~~~  Cumulative Processing ~~~~~~~~~

int ReadInitialCumulative(map *time_map, char *filename);

int ReadACP(map *st_map);

void ReadCumulativeFileLine(map *cum_map, char *acp_line);

int ReadAcpPipeLine(map *stmap, char *acp_line);

void CalculateCumulative(Student *stud_arr, int stud_arr_len, map *cum_map);

#endif
```

5.5 src/server/Data.c File Reference

```c
#include "Data.h"
```

Include dependency graph for Data.c:



Variables

- char ∗ Data_IDs [DATA_NUM_RECORDS]

  Data structures and constants.
- char ∗ Data_Names [DATA_NUM_RECORDS]

5.5.1 Variable Documentation

5.5.1.1 Data_IDs

```
char* Data_IDs[DATA_NUM_RECORDS]
```

Initial value:

```
= {
    "chen",
    "bea1389",
    "bol4559",
    "cal6258",
    "kre5277",
    "lon1150",
    "mas9309",
    "mes08346",
    "mil7233",
    "nef9476",
    "Nov-88",
    "pan9725",
    "rac3146",
    "rub4133",
    "shr5683",
    "vay3083",
    "yos2327"}
```

Data structures and constants.

Definition at line 6 of file Data.c.

5.5.1.2 Data_Names

```
char* Data_Names[DATA_NUM_RECORDS]
```

Initial value:

```
= {
    "Weifeng Chen",
    "Christian Beatty",
    "Emily Bolles",
    "Cameron Calhoun",
    "Ty Kress",
    "Cody Long",
    "Caleb Massey",
    "Christian Messmer",
    "Karl Miller",
    "Jeremiah Neff",
    "Kaitlyn Novacek",
    "Joshua Panaro",
    "Caleb Rachocki",
    "Caleb Ruby",
    "Paul Shriner",
    "Alan Vayansky",
    "Assefa Ayalew Yoseph"}
```

Constant, all user's names.

Definition at line 25 of file Data.c.

5.6 Data.c

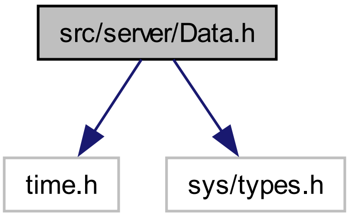
Go to the documentation of this file.

```

#include "Data.h"

char *Data_IDs[DATA_NUM_RECORDS] = {
    "chen",
    "bea1389",
    "bol4559",
    "cal6258",
    "kre5277",
    "lon1150",
    "mas9309",
    "mes08346",
    "mil7233",
    "nef9476",
    "Nov-88",
    "pan9725",
    "rac3146",
    "rub4133",
    "shr5683",
    "vay3083",
    "yos2327"};

char *Data_Names[DATA_NUM_RECORDS] = {
    "Weifeng Chen",
    "Christian Beatty",
    "Emily Bolles",
    "Cameron Calhoun",
    "Ty Kress",
    "Cody Long",
    "Caleb Massey",
    "Christian Messmer",
    "Karl Miller",
    "Jeremiah Neff",
    "Kaitlyn Novacek",
    "Joshua Panaro",
    "Caleb Rachocki",
    "Caleb Ruby",
    "Paul Shriner",
    "Alan Vayansky",
    "Assefa Ayalew Yoseph"};
```

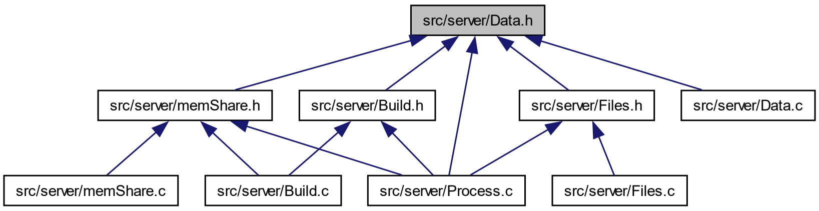
5.7 src/server/Data.h File Reference

```
#include <time.h>
#include <sys/types.h>
```

Include dependency graph for Data.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct Student

  The student data type.

Macros

- #define DATA_NUM_RECORDS 17

  Declarations of types and macros.
- #define DATA_ID_MAX_LENGTH 9
- #define DATA_NAME_MAX_LENGTH 21
- #define DATA_SIZE 56

Variables

- char ∗ Data_IDs [ ]

  Data structures and constants.
- char ∗ Data_Names [ ]

5.7.1 Macro Definition Documentation

5.7.1.1 DATA_ID_MAX_LENGTH

```
#define DATA_ID_MAX_LENGTH 9
```

The amount of memory (bytes) required to be allocated for the ID field. Equal to the longest name in Data_IDs, "mes08346", plus the null terminator

Definition at line 17 of file Data.h.

5.7.1.2 DATA_NAME_MAX_LENGTH

```
#define DATA_NAME_MAX_LENGTH 21
```

The amount of memory (bytes) required to be allocated for the Name field. Equal to the longest name in Data_↩ Names, "Assefa Ayalew Yoseph", plus the null terminator

Definition at line 22 of file Data.h.

5.7.1.3 DATA_NUM_RECORDS

```
#define DATA_NUM_RECORDS 17
```

Declarations of types and macros.

The total count of records.

Definition at line 12 of file Data.h.

5.7.1.4 DATA_SIZE

```
#define DATA_SIZE 56
```

The size of one student record; the result of sizeof(Student).

Definition at line 33 of file Data.h.

5.7.2 Variable Documentation

5.7.2.1 Data_IDs

```
char* Data_IDs[ ]  [extern]
```

Data structures and constants.

Definition at line 6 of file Data.c.

5.7.2.2 Data_Names

```
char* Data_Names[ ]  [extern]
```

Constant, all user's names.

Definition at line 25 of file Data.c.

5.8 Data.h

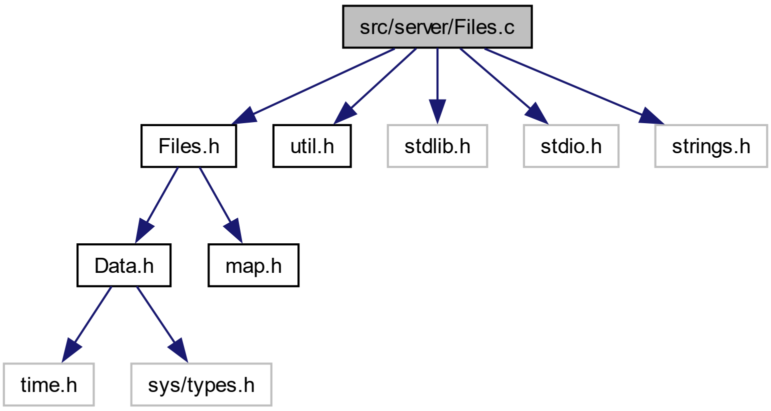
Go to the documentation of this file.

```
#ifndef Data_h
#define Data_h
#include <time.h>
#include <sys/types.h>

#define DATA_NUM_RECORDS 17
#define DATA_ID_MAX_LENGTH 9
#define DATA_NAME_MAX_LENGTH 21

/* Constant, all user IDs.  */
extern char *Data_IDs[];

extern char *Data_Names[];

#define DATA_SIZE 56

typedef struct
{
    char userID[DATA_ID_MAX_LENGTH];
    char fullName[DATA_NAME_MAX_LENGTH];
    short age;
    float gpa;
    short active;
    time_t lastLogin;
    int loginDuration;
} Student;

#endif
```

5.9 src/server/Files.c File Reference

```
#include "Files.h"
#include "util.h"
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
```

Include dependency graph for Files.c:



Functions

- short FileExists (char *file_name_to_check)

Declarations of functions that operate on files..
- int CreateInitialUserDataFile (char ∗file_name, char ∗∗id_list, int id_list_len)

  Creates the initial user data file. This should be called only the first time the program runs, if it doesn't exist.
- int FillStudentMapFromFile (map ∗student_map, char ∗file_name, char ∗∗id_list, int id_list_len)

  Fills the student map with data from the file. It gets age, gpa, and lastLogin from this file.
- int WriteStudentArrayToFile (Student ∗students, int arr_len, char ∗file_name)

  Writes the student array to the file.
- int CreateInitialCumulativeFile (char ∗file_name)

5.9.1 Function Documentation

5.9.1.1 CreateInitialCumulativeFile()

```
int CreateInitialCumulativeFile (
            char ∗ file_name )
```

Creates the initial cumulative login time file.

It will hold the result of running 'ac -p'.

Parameters

file_name	The name of the file to created. EG STATIC_USER_CUMULATIVE_FILE

Warning

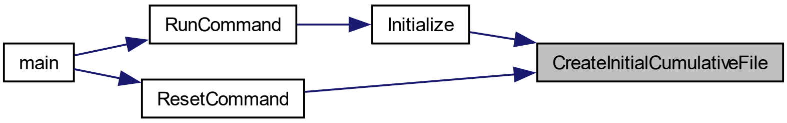
This file should already be validated to not exist.

Returns

0 if succesful, -1 if the file couldn't be opened, -2 if the pipe couldn't be opened, otherwise an error code.

Definition at line 94 of file Files.c.

Here is the caller graph for this function:



5.9.1.2 CreateInitialUserDataFile()

```
int CreateInitialUserDataFile (
            char * file_name,
            char ** id_list,
            int id_list_len )
```

Creates the initial user data file. This should be called only the first time the program runs, if it doesn't exist.

Parameters

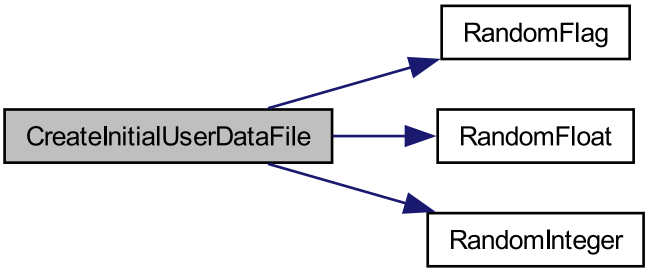
file_name	The file name to create.
id_list	An array containing the IDs. Eg. "Data_IDs" from Data.h
id_list_len	The length of the id_list. Eg. "DATA_NUM_RECORDS" from Data.h

Returns

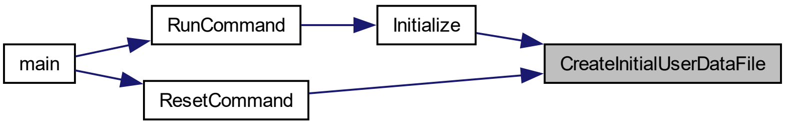
A 0 if the operation was succesful, otherwise nonzero.

Definition at line 25 of file Files.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.1.3 FileExists()

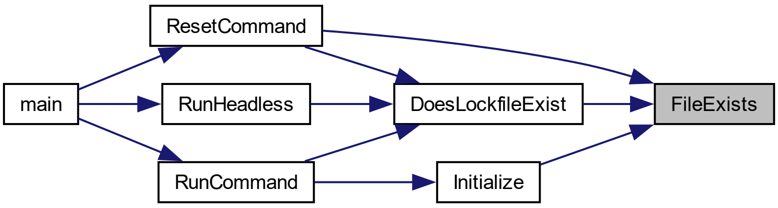
```
short FileExists (
             char * file_name_to_check )
```

Declarations of functions that operate on files..

Determines whether a file exists.

Definition at line 10 of file Files.c.

Here is the caller graph for this function:



5.9.1.4 FillStudentMapFromFile()

```
int FillStudentMapFromFile (
             map * student_map,
             char * file_name,
             char ** id_list,
             int id_list_len )
```

Fills the student map with data from the file. It gets age, gpa, and lastLogin from this file.

Parameters

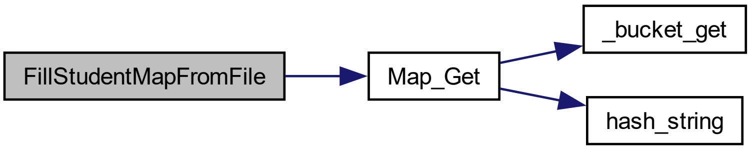
student_map	The map of student structs to be populated from the login.txt file
file_name	The name of the login.txt file.
id_list	An array containing the IDs. Eg. "Data_IDs" from Data.h
id_list_len	The length of the id_list. Eg. "DATA_NUM_RECORDS" from Data.h

Returns

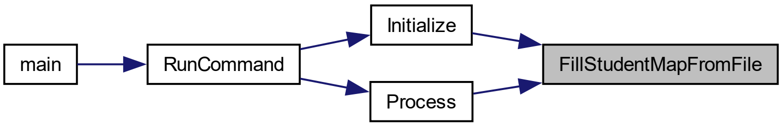
0 if succesful, 1 if there was an error.

Definition at line 51 of file Files.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.9.1.5 WriteStudentArrayToFile()

```
int WriteStudentArrayToFile (
                Student * students,
                int arr_len,
                char * file_name )
```

Writes the student array to the file.

Parameters

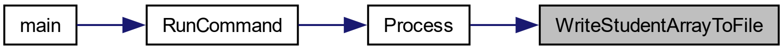
students	A pointer to the student array that will be read into the file.
arr_len	The length of the students array. e.g. DATA_NUM_RECORDS from Data.h.
file_name	The file name to write.

Returns

A 0 if the operation was succesful, otherwise a nonzero.

Definition at line 78 of file Files.c.

Here is the caller graph for this function:



5.10 Files.c

Go to the documentation of this file.

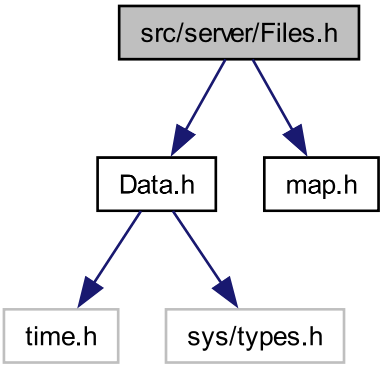
```

#include "Files.h"
#include "util.h"
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>

short FileExists(char *file_name_to_check)
{
    FILE *file = fopen(file_name_to_check, "r");
    short result = 1;
    if (file == NULL)
    {
        result = 0;
    }
    else
    {
        fclose(file);
    }
    return result;
}

int CreateInitialUserDataFile(char *file_name, char **id_list, int id_list_len)
{
    FILE *file = fopen(file_name, "w");
    if (file == NULL)
    {
        return -1;
    }
    int i;
    for (i = 0; i < id_list_len; i++)
    {
        int rand_age = RandomInteger(18, 22);
        float gpa;
        if (RandomFlag(0.42))
        {
            gpa = 4.0; // 42% of the time, make the GPA 4.0
        }
        else
        {
            gpa = RandomFloat(2.5, 4.0);
        }
        fprintf(file, "%s\t%d\t%.2f\t%d\n", id_list[i], rand_age, gpa, 0);
    }
    fclose(file);
    return 0;
}

int FillStudentMapFromFile(map *student_map, char *file_name, char **id_list, int id_list_len)
{
    FILE *file = fopen(file_name, "r");
    if (file == NULL)
    {
        return -1;
    }
    // id buffer
    char user_id[9];
    int age;
    float gpa;
    long time;
    while (fscanf(file, "%9s\t%d\t%f\t%ld", user_id, &age, &gpa, &time) == 4)
    {
        map_result result = Map_Get(student_map, user_id);
        if (result.found == 0)
```

```c
        {
            continue;
        }
        ((Student *)result.data)->age = age;
        ((Student *)result.data)->gpa = gpa;
        ((Student *)result.data)->lastLogin = time;
    }
    fclose(file);
    return 0;
}

int WriteStudentArrayToFile(Student *students, int arr_len, char *file_name)
{
    FILE *file = fopen(file_name, "w");
    if (file == NULL)
    {
        return -1;
    }
    int i;
    for (i = 0; i < arr_len; i++)
    {
        fprintf(file, "%s\t%d\t%.2f\t%ld\n", students[i].userID, students[i].age, students[i].gpa, students[i].lastLogin);
    }
    fclose(file);
    return 0;
}

int CreateInitialCumulativeFile(char *file_name)
{
    FILE *file = fopen(file_name, "w");
    if (file == NULL)
    {
        return -1;
    }
    FILE *pipe = popen("ac -p", "r");
    if (pipe == NULL)
    {
        fclose(file);
        return -2;
    }

    char line[100];
    while (fgets(line, sizeof(line), pipe) != NULL)
    {
        fputs(line, file);
    }
    pclose(pipe);
    fclose(file);
    return 0;
}
```

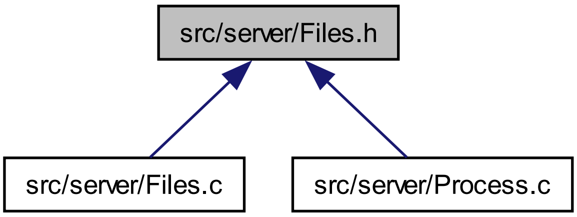
5.11 src/server/Files.h File Reference

```c
#include "Data.h"
#include "map.h"
```

Include dependency graph for Files.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define STATIC_USER_DATA_FILE "static-user-data.txt"

  Definitions for functions that operate on files.
- #define STATIC_USER_CUMULATIVE_FILE "static-user-cumulative-start.txt"

Functions

- short FileExists (char ∗file_name_to_check)

  Determines whether a file exists.
- int CreateInitialUserDataFile (char ∗file_name, char ∗∗id_list, int id_list_len)

  Creates the initial user data file. This should be called only the first time the program runs, if it doesn't exist.
- int WriteStudentArrayToFile (Student ∗students, int arr_len, char ∗file_name)

  Writes the student array to the file.
- int FillStudentMapFromFile (map ∗student_map, char ∗file_name, char ∗∗id_list, int id_list_len)

  Fills the student map with data from the file. It gets age, gpa, and lastLogin from this file.
- int CreateInitialCumulativeFile (char ∗file_name)

5.11.1 Macro Definition Documentation

5.11.1.1 STATIC_USER_CUMULATIVE_FILE

```
#define STATIC_USER_CUMULATIVE_FILE "static-user-cumulative-start.txt"
```

File name for the text file that will store the cumulative login time for each user at the point in time when it was created.

The values in this file are subtracted from the result of running 'ac -p' later to get the cumulative time each user was logged in since the server started.

Definition at line 38 of file Files.h.

5.11.1.2 STATIC_USER_DATA_FILE

```
#define STATIC_USER_DATA_FILE "static-user-data.txt"
```

Definitions for functions that operate on files.

Some program data needs to be stored in files, to preserve it in the case of early termination.

There are two files that are created if they don't exist when the program is first run. One has userIDs linked to their age, gpa, and last login time. The other has userIDs linked to the cumulative login time as determined by `ac -p` in order to determine the time logged in since program first ran. File name for the text file that will store user data, namely, the age, gpa, and last login time.

Note

> Each line contain in the created file contains:
>
> (1) The ID from the students array, where the `line # - 1` == the index of the students array
>
> (2) A tab character
>
> (3) A random int between 18 and 22, for the age.
>
> (4) A tab character
>
> (5) A random float between 2.5 and 4.0, for the GPA.
>
> (6) A tab character.
>
> (7) A 0 (representing the last login time)
>
> (8) A newline.
>
> The order of entries in the file is the same as the order in the Data_IDs array from Data.c.

Definition at line 31 of file Files.h.

5.11.2 Function Documentation

5.11.2.1 CreateInitialCumulativeFile()

```
int CreateInitialCumulativeFile (
             char * file_name )
```

Creates the initial cumulative login time file.

It will hold the result of running 'ac -p'.

Parameters

file_name	The name of the file to created. EG STATIC_USER_CUMULATIVE_FILE

Warning

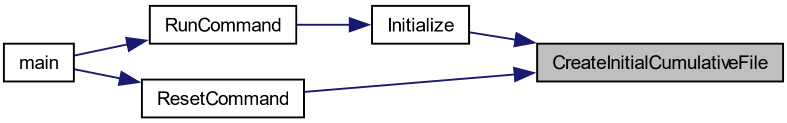
This file should already be validated to not exist.

Returns

0 if succesful, -1 if the file couldn't be opened, -2 if the pipe couldn't be opened, otherwise an error code.

Definition at line 94 of file Files.c.

Here is the caller graph for this function:



5.11.2.2 CreateInitialUserDataFile()

```
int CreateInitialUserDataFile (
             char * file_name,
             char ** id_list,
             int id_list_len )
```

Creates the initial user data file. This should be called only the first time the program runs, if it doesn't exist.

Parameters

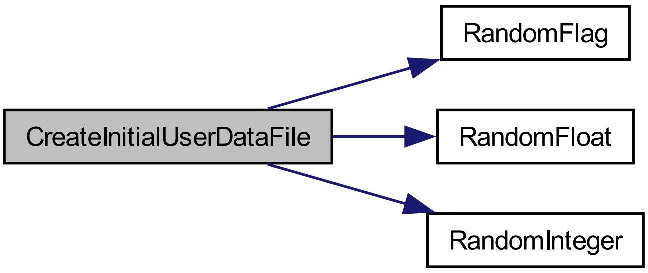
file_name	The file name to create.
id_list	An array containing the IDs. Eg. "Data_IDs" from Data.h
id_list_len	The length of the id_list. Eg. "DATA_NUM_RECORDS" from Data.h

Returns

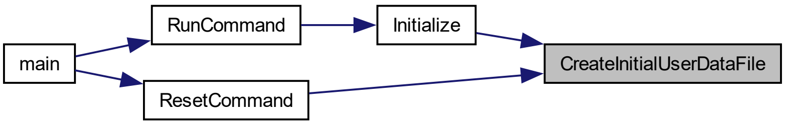
A 0 if the operation was succesful, otherwise nonzero.

Definition at line 25 of file Files.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.11.2.3 FileExists()

```
short FileExists (
            char * file_name_to_check )
```

Determines whether a file exists.

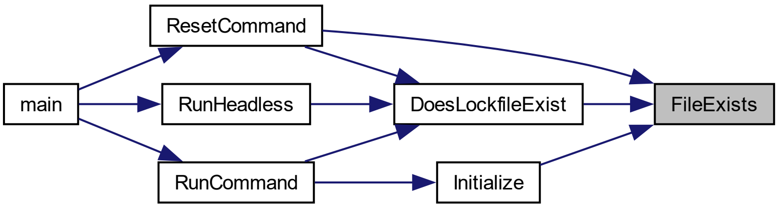
Returns

1 if it exists. 0 if it does not.

Determines whether a file exists.

Definition at line 10 of file Files.c.

Here is the caller graph for this function:



5.11.2.4 FillStudentMapFromFile()

```
int FillStudentMapFromFile (
            map * student_map,
            char * file_name,
            char ** id_list,
            int id_list_len )
```

Fills the student map with data from the file. It gets age, gpa, and lastLogin from this file.

Parameters

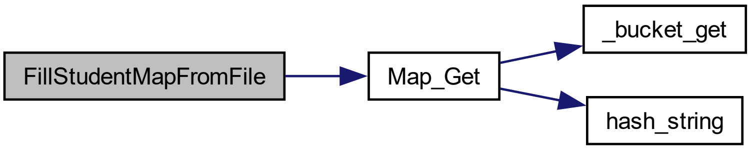
student_map	The map of student structs to be populated from the login.txt file
file_name	The name of the login.txt file.
id_list	An array containing the IDs. Eg. "Data_IDs" from Data.h
id_list_len	The length of the id_list. Eg. "DATA_NUM_RECORDS" from Data.h

Returns

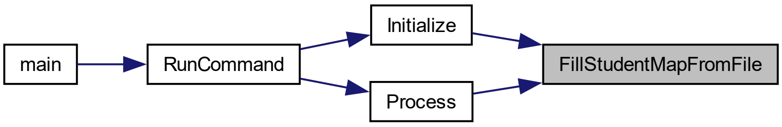
0 if succesful, 1 if there was an error.

Definition at line 51 of file Files.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.11.2.5 WriteStudentArrayToFile()

```
int WriteStudentArrayToFile (
            Student * students,
            int arr_len,
            char * file_name )
```

Writes the student array to the file.

Parameters

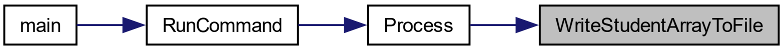
students	A pointer to the student array that will be read into the file.
arr_len	The length of the students array. e.g. DATA_NUM_RECORDS from Data.h.
file_name	The file name to write.

Returns

A 0 if the operation was succesful, otherwise a nonzero.

Definition at line 78 of file Files.c.

Here is the caller graph for this function:



5.12 Files.h

Go to the documentation of this file.

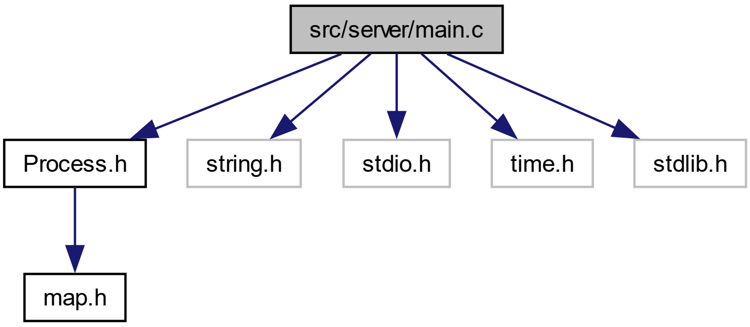
```
00001 #ifndef Files_H
00002 #define Files_H
```

```
#include "Data.h"
#include "map.h"

#define STATIC_USER_DATA_FILE "static-user-data.txt"

#define STATIC_USER_CUMULATIVE_FILE "static-user-cumulative-start.txt"

short FileExists(char *file_name_to_check);

int CreateInitialUserDataFile(char *file_name, char **id_list, int id_list_len);

int WriteStudentArrayToFile(Student *students, int arr_len, char *file_name);

int FillStudentMapFromFile(map *student_map, char *file_name, char **id_list, int id_list_len);

int CreateInitialCumulativeFile(char *file_name);

#endif
```

5.13 src/server/main.c File Reference

```
#include "Process.h"
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
```

Include dependency graph for main.c:



Functions

- int main (int argc, char **argv)

  Program entry.

5.13.1 Function Documentation

5.13.1.1 main()

```
int main (
            int argc,
            char ** argv )
```

Program entry.

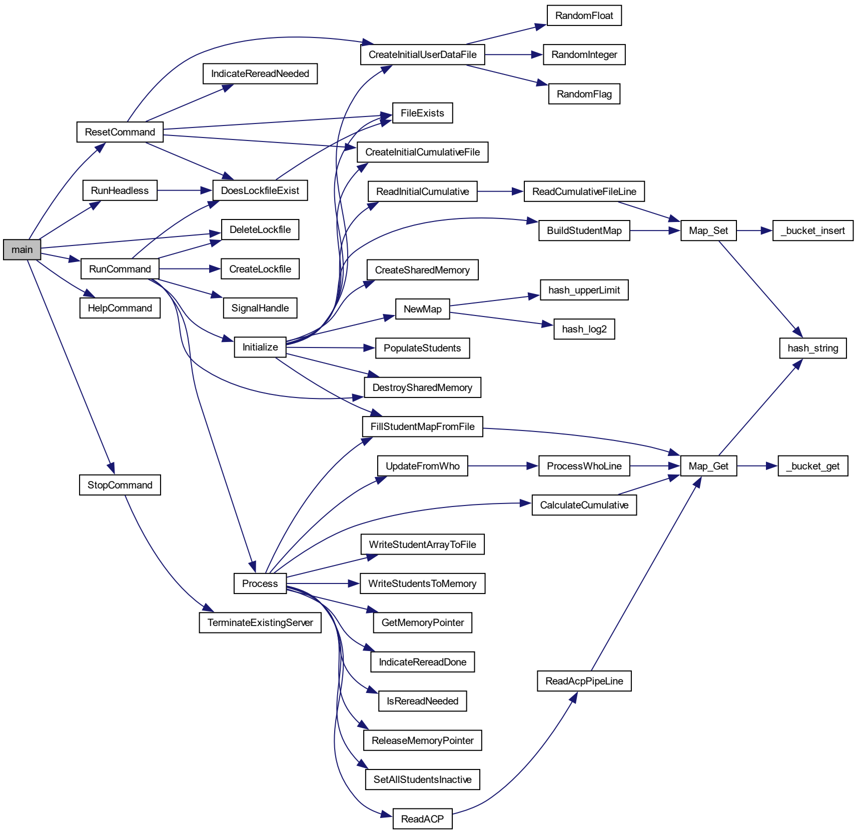
Parses arguments and calls the appropriate Process.h function.

Parameters

argc	The argument count.
argv	The argument values.

Definition at line 30 of file main.c.

Here is the call graph for this function:



5.14 main.c

Go to the documentation of this file.

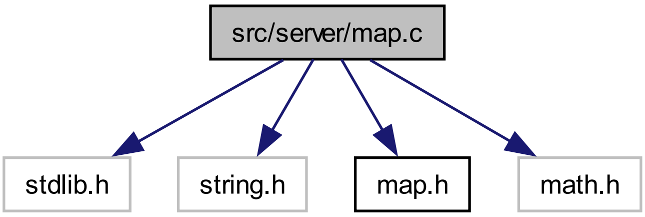
```
00001 #include "Process.h"
00002 #include <string.h>
```

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
    srand(time(NULL)); // seed the randomizer

    if (argc <= 1 || argc >= 3)
    {
        printf("You entered too few or many options!\n");
        HelpCommand();
    }
    else if (strcmp(argv[1], "help") == 0)
    {
        HelpCommand();
    }
    else if (strcmp(argv[1], "reset") == 0)
    {
        ResetCommand();
    }
    else if (strcmp(argv[1], "stop") == 0 || strcmp(argv[1], "end") == 0 || strcmp(argv[1], "close")
 == 0)
    {
        StopCommand();
    }
    else if (strcmp(argv[1], "headless") == 0)
    {
        RunHeadless(argv[0]);
    }
    else if (strcmp(argv[1], "run") == 0 || strcmp(argv[1], "start") == 0)
    {
        RunCommand();
    }
    else if (strcmp(argv[1], "delete-lockfile") == 0)
    { // an admin debug command; not meant to be called
        printf("Deleting lockfile.\n");
        DeleteLockfile();
    }
    else
    {
        printf("Unknown option!\n");
        HelpCommand();
    }
    return 0;
}
```

5.15 src/server/map.c File Reference

```
#include "stdlib.h"
#include "string.h"
#include "map.h"
#include "math.h"
```

Include dependency graph for map.c:



Functions

- int hash_log2 (int num_to_log)

  Definitions for functions relating to a hashmap data structure.
- int hash_upperLimit (int bitsize)

  This calculates what the actual capacity of the map will be. Given a result from hash_log2, it gets the maximum storable for that many bits. For example, hash_upperLimit(3) returns the maximum that 3 bits can hold, which is 8. hash_upperLimit(4) returns 16.
- int hash_string (int hash_table_size, char ∗string, int strlen)
- map ∗ NewMap (int capacity)
- void _bucket_insert (struct _map_bucket ∗bucket, char ∗key, void ∗value)
- void Map_Set (map ∗a_map, char ∗key, void ∗value)

  Sets a value in the map.
- void _bucket_get (struct _map_bucket ∗bucket, char ∗key, map_result ∗result)
- map_result Map_Get (map ∗a_map, char ∗key)

  Gets a value from the map. It will return a map_get_result describing whether it was succesful, and possibly containing the data sought, or NULL if it was unsuccesful.
- void _bucket_delete (struct _map_bucket ∗bucket, char ∗key, short free_it, map_result ∗result)
- map_result Map_Delete (map ∗a_map, char ∗key, short free_it)

  Deletes a key from the map. Returns a map_get_result describing whether the delete was succesful and containing the removed data, if extant.

Variables

- int char_ratio = (int)(sizeof(int) / sizeof(char))

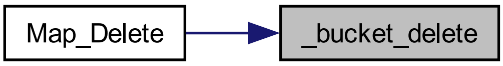
5.15.1 Function Documentation

5.15.1.1 _bucket_delete()

```
void _bucket_delete (
            struct _map_bucket ∗ bucket,
            char ∗ key,
            short free_it,
            map_result ∗ result )
```

Definition at line 123 of file map.c.

Here is the caller graph for this function:

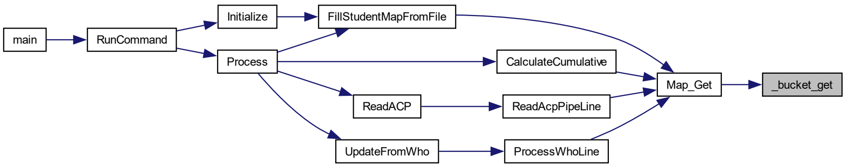


5.15.1.2 _bucket_get()

```
void _bucket_get (
            struct _map_bucket * bucket,
            char * key,
            map_result * result )
```

Definition at line 91 of file map.c.

Here is the caller graph for this function:

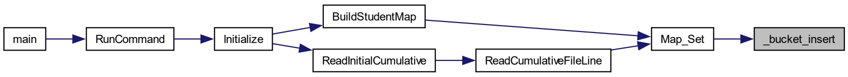


5.15.1.3 _bucket_insert()

```
void _bucket_insert (
            struct _map_bucket * bucket,
            char * key,
            void * value )
```

Definition at line 63 of file map.c.

Here is the caller graph for this function:



5.15.1.4 hash_log2()

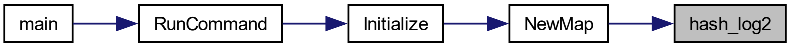
```
int hash_log2 (
            int num_to_log )
```

Definitions for functions relating to a hashmap data structure.

Definitions for functions that operate on a hash map data structure. Karl's take on a simple hashmap map structure, which maps strings to void pointers. You can use casting to convert the void pointers into most of whatever else is needed.

Definition at line 9 of file map.c.

Here is the caller graph for this function:



5.15.1.5 hash_string()

```
int hash_string (
            int hash_table_capacity,
            char * string,
            int strlen )
```

Uses some clever, prime-number-multiplication, ORing, and bitwise operations to generate a number than, when modulused with the hash_table_size, will produce numbers ('buckets') of even distribution, to minimize the number of collisions. This function contains the meat of the hashing algorithm; it converts a key-string to an array index.

See also

http://isthe.com/chongo/tech/comp/fnv/

Parameters

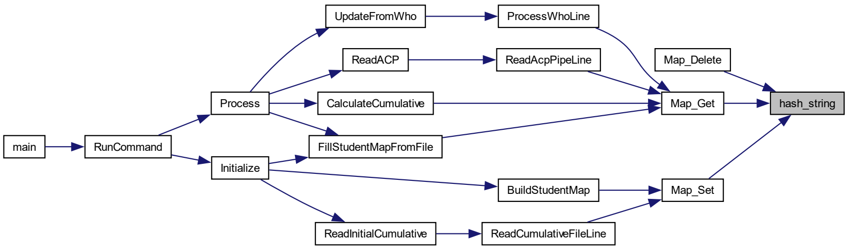
hash_table_capacity	The number of buckets this table holds.
string	The key to hash.
strlen	The length of the key.

Returns

The index of the bucket that should be used.

Definition at line 30 of file map.c.

Here is the caller graph for this function:



5.15.1.6 hash_upperLimit()

```
int hash_upperLimit (
            int bitsize )
```

This calculates what the actual capacity of the map will be. Given a result from hash_log2, it gets the maximum storable for that many bits. For example, hash_upperLimit(3) returns the maximum that 3 bits can hold, which is 8. hash_upperLimit(4) returns 16.

Parameters

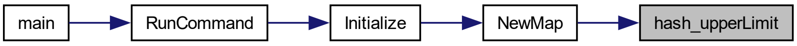
bitsize	The number of bits to calculate the max from.

Returns

The max value that number of bits can hold.

Definition at line 22 of file map.c.

Here is the caller graph for this function:



5.15.1.7 Map_Delete()

```
map_result Map_Delete (
            map * a_map,
            char * key,
            short free_it )
```

Deletes a key from the map. Returns a map_get_result describing whether the delete was succesful and containing the removed data, if extant.

Parameters

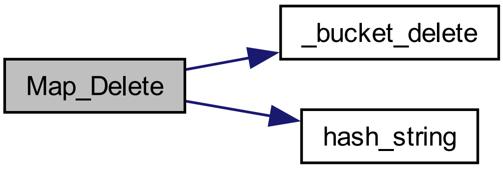
map	The map to delete the key from.
key	The key to delete.
free↩_it	Whether to call free() on the underlying data.

Returns

A map_get_result with the data that was removed.

Definition at line 149 of file map.c.

Here is the call graph for this function:



5.15.1.8 Map_Get()

```
map_result Map_Get (
            map * a_map,
            char * key )
```

Gets a value from the map. It will return a map_get_result describing whether it was succesful, and possibly containing the data sought, or NULL if it was unsuccesful.

Parameters

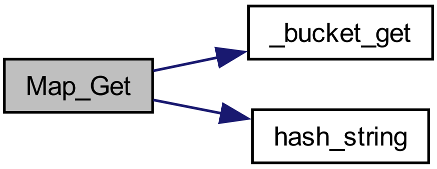
map	The map to retrieve from.
key	The key of the item.

Returns

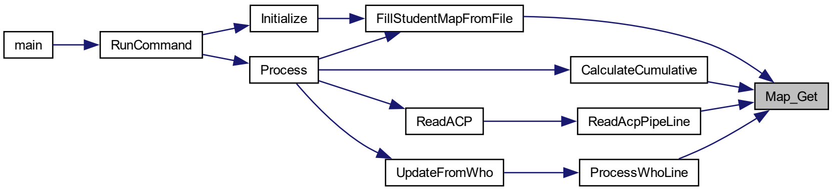
A map_get_result containing the sought data.

Definition at line 114 of file map.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.15.1.9 Map_Set()

```
void Map_Set (
            map * a_map,
            char * key,
            void * value )
```

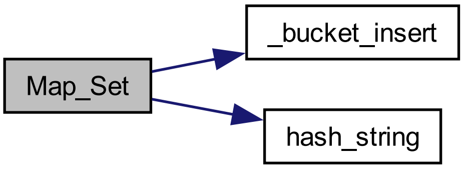
Sets a value in the map.

Parameters

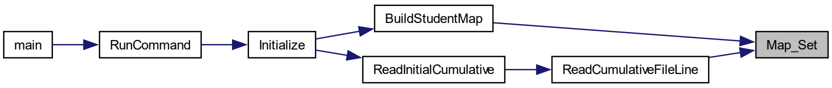
map	The map to set a key in.
key	The key to use.
keylen	The length of the key.
value	The pointer to the data stored at that location.

Definition at line 84 of file map.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.15.1.10 NewMap()

```
map * NewMap (
            int capacity )
```

Creates a new map. The map capacity will be a power of 2 that is large enough to contain the estimated size.

Parameters

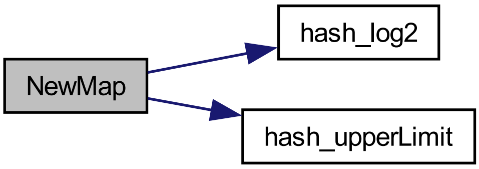
capacity	The estimated required capacity of the map.

Returns

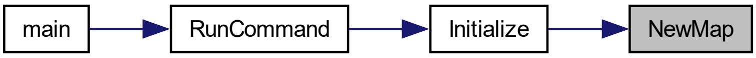
A pointer to the heap allocated map.

Definition at line 45 of file map.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.15.2 Variable Documentation

5.15.2.1 char_ratio

```
int char_ratio = (int)(sizeof(int) / sizeof(char))
```

Definition at line 27 of file map.c.

5.16 map.c

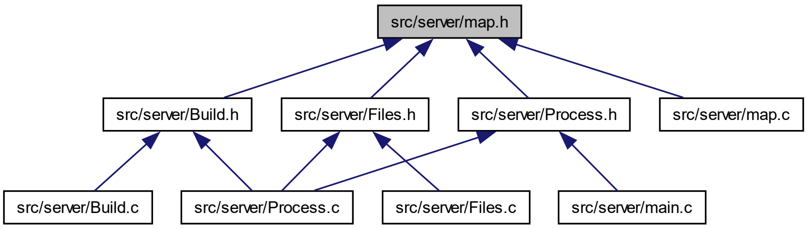
Go to the documentation of this file.

```
00001
00004 #include "stdlib.h"
00005 #include "string.h"
00006 #include "map.h"
00007 #include "math.h"
00008
00009 int hash_log2(int num_to_log)
00010 {
00011     int t = 1;
00012     int i = 0;
00013     do
00014     {
00015         num_to_log = num_to_log & ~t;
00016         t = t « 1;
00017         i++;
00018     } while (num_to_log > 0);
00019     return i;
```

```
}

int hash_upperLimit(int bitsize)
{
    return 1 « bitsize;
}

int char_ratio = (int)(sizeof(int) / sizeof(char));

// Modified some stuff from :  http://isthe.com/chongo/tech/comp/fnv/
int hash_string(int hash_table_size, char *string, int strlen)
{
    int i, hash = 2166136261;
    for (i = 0; i < strlen; i += 1)
    {
        hash *= 16777619;
        hash ^= string[i];
    }
    if (hash < 0)
    {
        hash *= -1;
    }
    return hash % (hash_table_size - 1) + 1;
}

map *NewMap(int capacity)
{
    int log2 = hash_log2(capacity);
    int capac = hash_upperLimit(log2);
    int sz = sizeof(struct _map_bucket) * capac;
    struct _map_bucket *buckets = malloc(sz);
    memset(buckets, 0, sz);
    int i;
    for (i = 0; i < capac; i++)
    {
        buckets[i] = (struct _map_bucket){NULL, NULL, NULL};
    }
    map newm = (map){capac, buckets};
    map *map_p = malloc(sizeof(map));
    *map_p = newm;
    return map_p;
}

void _bucket_insert(struct _map_bucket *bucket, char *key, void *value)
{
    struct _map_bucket *check = bucket;
    while (check->key != NULL)
    {
        if (strcmp(check->key, key) == 0)
        {
            check->data = value;
            return;
        }
        if (check->next == NULL)
        {
            check->next = malloc(sizeof(struct _map_bucket));
            *(check->next) = (struct _map_bucket){NULL, NULL, NULL};
        }
        check = check->next;
    }
    check->key = key;
    check->data = value;
}

void Map_Set(map *a_map, char *key, void *value)
{
    int keyl = (int)strlen(key);
    int hash = hash_string(a_map->size, key, keyl);
    _bucket_insert(&(a_map->buckets[hash]), key, value);
}

void _bucket_get(struct _map_bucket *bucket, char *key, map_result *result)
{
    struct _map_bucket *check = bucket;
    while (check->key != NULL)
    {
        if (strcmp(check->key, key) == 0)
        {
            result->found = 1;
            result->data = check->data;
            return;
        }
        else if (check->next != NULL)
        {
            check = check->next;
        }
        else
        {
            result->found = 0;
            break;
        }
    }
}

map_result Map_Get(map *a_map, char *key)
{
    map_result res = (map_result){0, NULL};
    int keyl = (int)strlen(key);
    int hash = hash_string(a_map->size, key, keyl);
    _bucket_get(&(a_map->buckets[hash]), key, &res);
    return res;
}

void _bucket_delete(struct _map_bucket *bucket, char *key, short free_it, map_result *result)
{
    struct _map_bucket *last = bucket;
    struct _map_bucket *next = bucket->next;
    while (next != NULL)
    {
        if (strcmp(next->key, key) == 0)
        {
            result->found = 1;
            result->data = next->data;
            if (free_it)
            {
                free(next->data);
                result->data = NULL;
            }
            last->next = next->next;
            free(next);
        }
        else
        {
            last = next;
            next = next->next;
        }
    }
}

map_result Map_Delete(map *a_map, char *key, short free_it)
{
    map_result res = (map_result){0, NULL};
    int keyl = (int)strlen(key);
    int hash = hash_string(a_map->size, key, keyl);

    struct _map_bucket top = a_map->buckets[hash];
    if (top.key == NULL)
    {
        return res;
    }
    if (strcmp(top.key, key) == 0)
    {
        res.found = 1;
        res.data = top.data;
        if (free_it)
        {
            free(top.data);
            res.data = NULL;
        }
        if (top.next != NULL)
        {
            a_map->buckets[hash] = *(top.next);
            free(top.next);
        }
        else
        {
            a_map->buckets[hash] = (struct _map_bucket){NULL, NULL, NULL};
        }
        return res;
    }
    if (top.next == NULL)
    {
        return res;
    }
    _bucket_delete(&(a_map->buckets[hash]), key, free_it, &res);

    return res;
}
```

5.17 src/server/map.h File Reference

This graph shows which files directly or indirectly include this file:



Data Structures

- struct map

  A map. Stores key-value pairs for near constant lookup and insertion time.
- struct map_result

  The result of a map retrieval.

Functions

- int hash_log2 (int number_to_log)

  Definitions for functions that operate on a hash map data structure. Karl's take on a simple hashmap map structure, which maps strings to void pointers. You can use casting to convert the void pointers into most of whatever else is needed.
- int hash_string (int hash_table_capacity, char ∗string, int strlen)
- int hash_upperLimit (int bitsize)

  This calculates what the actual capacity of the map will be. Given a result from hash_log2, it gets the maximum storable for that many bits. For example, hash_upperLimit(3) returns the maximum that 3 bits can hold, which is 8. hash_upperLimit(4) returns 16.
- map ∗ NewMap (int capacity)
- void Map_Set (map ∗a_map, char ∗key, void ∗value)

  Sets a value in the map.
- map_result Map_Get (map ∗a_map, char ∗key)

  Gets a value from the map. It will return a map_get_result describing whether it was succesful, and possibly containing the data sought, or NULL if it was unsuccesful.
- map_result Map_Delete (map ∗a_map, char ∗key, short free_it)

  Deletes a key from the map. Returns a map_get_result describing whether the delete was succesful and containing the removed data, if extant.

5.17.1 Function Documentation

5.17.1.1 hash_log2()

```
int hash_log2 (
              int num_to_log )
```

Definitions for functions that operate on a hash map data structure. Karl's take on a simple hashmap map structure, which maps strings to void pointers. You can use casting to convert the void pointers into most of whatever else is needed.

Example usage, casting an into the data part of the map.

```
int myfunc() {
     map *mymap = NewMap(100);
     Map_Set(mymap, "age", (void*)55);
     map_result result = Map_Get(mymap, "age");
     int age;
     if(result.found) {
         age = (int) map_result.data;
     }
}
```

Note, with this simple implementation, the map cannot change its capacity. A change to its capacity would change the hashing.

Ultimately there are really only three things you need to do with the map.

Initialize it, with some capacity larger than you will use. EG map ∗ mymap = NewMap(100). The bigger it is, the fewer collisions (which are pretty rare anyway).

Set some values in it. Eg Map_Set(mymap, "key", &value);

You can cast numbers to void pointers to put them in the map, or you can use the pointers as references to, for example, strings malloced somewhere.

Get some values from it. Eg void∗ myval = Map_Get(mymap, "key");

Delete some values from it. For example Map_Delete(mymap, "key", 0);

Note that the last parameter, 'free it', tells the map whether it should call 'free' on the underlying data in memory. If this is 1, and the underlying data is not a reference to a malloced part of the heap, errors will result.

---— Some Improvements.

1. Map can free on delete. We could have a Map_Set(map, key, char ∗) that will automatically malloc a string on set, to simplify string to string maps.
2. We could have convenience methods that auto-cast for various types of data. Eg. Map_GetTime...

Get's a log2 ceiling. Eg, hash_log2(5) == 3.

Parameters

number_to_log	The number to calculate the log of.

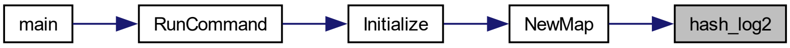
Returns

The log ceiling; eg, the lowest exponent to raise 2 with which would yield a number greater or equal to number_to_log.

Definitions for functions that operate on a hash map data structure. Karl's take on a simple hashmap map structure, which maps strings to void pointers. You can use casting to convert the void pointers into most of whatever else is needed.

Definition at line 9 of file map.c.

Here is the caller graph for this function:



5.17.1.2 hash_string()

```
int hash_string (
             int hash_table_capacity,
             char * string,
             int strlen )
```

Uses some clever, prime-number-multiplication, ORing, and bitwise operations to generate a number than, when modulused with the hash_table_size, will produce numbers ('buckets') of even distribution, to minimize the number of collisions. This function contains the meat of the hashing algorithm; it converts a key-string to an array index.

See also

http://isthe.com/chongo/tech/comp/fnv/

Parameters

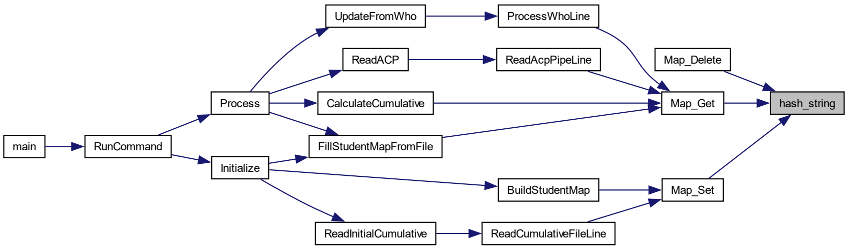
hash_table_capacity	The number of buckets this table holds.
string	The key to hash.
strlen	The length of the key.

Returns

The index of the bucket that should be used.

Definition at line 30 of file map.c.

Here is the caller graph for this function:



5.17.1.3 hash_upperLimit()

```
int hash_upperLimit (
              int bitsize )
```

This calculates what the actual capacity of the map will be. Given a result from hash_log2, it gets the maximum storable for that many bits. For example, hash_upperLimit(3) returns the maximum that 3 bits can hold, which is 8. hash_upperLimit(4) returns 16.

Parameters

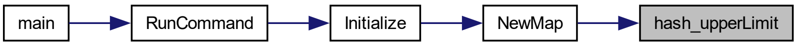
bitsize	The number of bits to calculate the max from.

Returns

The max value that number of bits can hold.

Definition at line 22 of file map.c.

Here is the caller graph for this function:



5.17.1.4 Map_Delete()

```
map_result Map_Delete (
            map * a_map,
            char * key,
            short free_it )
```

Deletes a key from the map. Returns a map_get_result describing whether the delete was succesful and containing the removed data, if extant.

Parameters

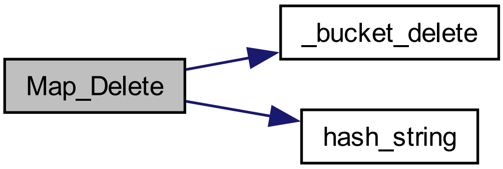
map	The map to delete the key from.
key	The key to delete.
free↩_it	Whether to call free() on the underlying data.

Returns

A map_get_result with the data that was removed.

Definition at line 149 of file map.c.

Here is the call graph for this function:



5.17.1.5 Map_Get()

```
map_result Map_Get (
            map * a_map,
            char * key )
```

Gets a value from the map. It will return a map_get_result describing whether it was succesful, and possibly containing the data sought, or NULL if it was unsuccesful.

Parameters

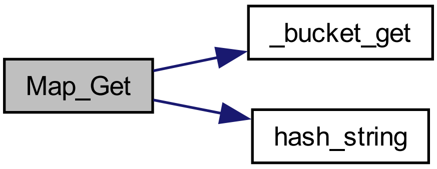
map	The map to retrieve from.
key	The key of the item.

Returns

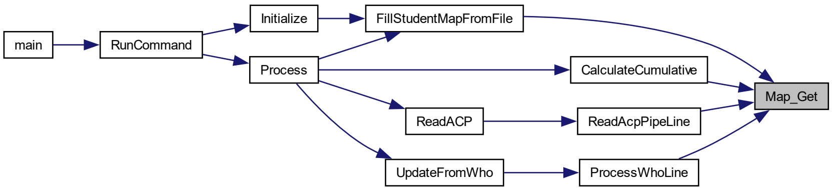
A map_get_result containing the sought data.

Definition at line 114 of file map.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.17.1.6 Map_Set()

```
void Map_Set (
             map * a_map,
             char * key,
             void * value )
```

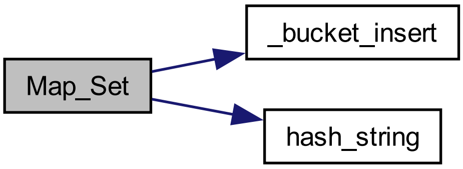
Sets a value in the map.

Parameters

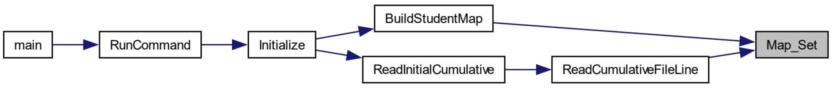
map	The map to set a key in.
key	The key to use.
keylen	The length of the key.
value	The pointer to the data stored at that location.

Definition at line 84 of file map.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.17.1.7 NewMap()

```
map * NewMap (
              int capacity )
```

Creates a new map. The map capacity will be a power of 2 that is large enough to contain the estimated size.

Parameters

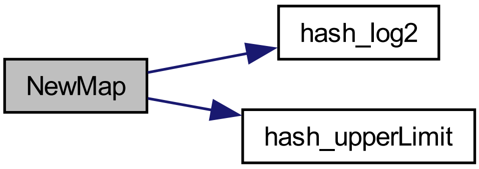
capacity	The estimated required capacity of the map.

Returns

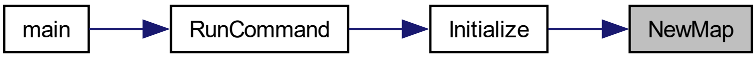
A pointer to the heap allocated map.

Definition at line 45 of file map.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.18 map.h

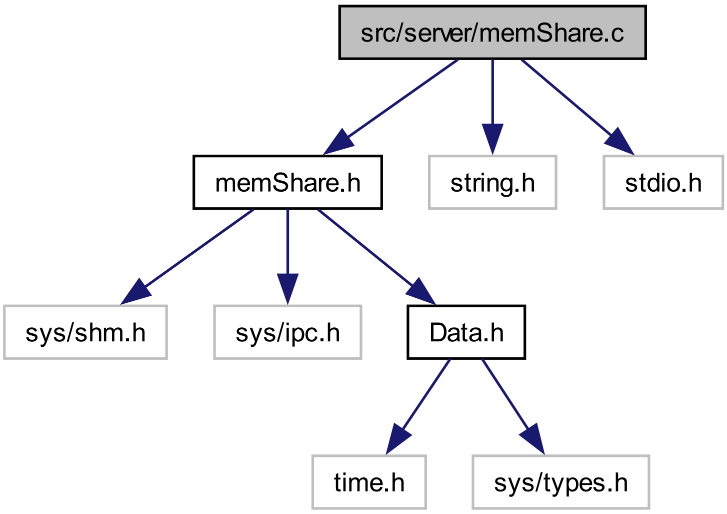
Go to the documentation of this file.

```
#ifndef map_h
#define map_h

// ----------------------------
//         Hashing Math
// ----------------------------

int hash_log2(int number_to_log);

int hash_string(int hash_table_capacity, char *string, int strlen);

int hash_upperLimit(int bitsize);

// ------------------------------------
//         General Map Operations
// ------------------------------------

struct _map_bucket
{
    // The key associated with this bucket.
    char *key;
    // The data this bucket holds.
    void *data;
    // The next node in this linked list, or NULL if it is a leaf.
    struct _map_bucket *next;
};

typedef struct
{
    // The number of base buckets in this map.
    int size;
    // The buckets for this map.
    struct _map_bucket *buckets;
} map;

typedef struct
{
    // 1 if succesfully found.  0 if not found.
    short found;
    // The data linked with that key; indeterminate if found == 0.
    void *data;
} map_result;

map *NewMap(int capacity);

void Map_Set(map *a_map, char *key, void *value);

map_result Map_Get(map *a_map, char *key);

map_result Map_Delete(map *a_map, char *key, short free_it);

#endif
```

5.19 src/server/memShare.c File Reference

```
#include "memShare.h"
#include <string.h>
#include <stdio.h>
```

Include dependency graph for memShare.c:



Functions

- int CreateSharedMemory ()

  Definitions for functions that operate on a shared memory segment.
- int DestroySharedMemory ()
- void ∗ GetMemoryPointer (int shared_mem_id)
- int ReleaseMemoryPointer (void ∗shmaddr)

5.19.1 Function Documentation

5.19.1.1 CreateSharedMemory()

```
int CreateSharedMemory ( )
```

Definitions for functions that operate on a shared memory segment.

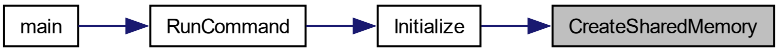
CreateSharedMemory retrieves a shared memory ID that can be used to access or delete shared memory.

Returns

A shared memory ID that can be used with other 'shm' commands to access shared memory.

Definition at line 12 of file memShare.c.

Here is the caller graph for this function:



5.19.1.2 DestroySharedMemory()

```
int DestroySharedMemory ( )
```

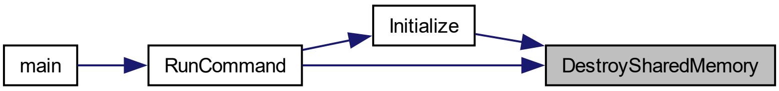
Flags the shared memory segment for deallocation. Returns the result of that operation.

Returns

0 if succesful. 1 if not succesful. Errno will be set.

Definition at line 17 of file memShare.c.

Here is the caller graph for this function:



5.19.1.3 GetMemoryPointer()

```
void * GetMemoryPointer (
            int shared_mem_id )
```

"Attaches" to the shared memory, returning a memory pointer to the shared memory.

Calls 'shmat(shared_mem_id, NULL, 0)`;

Parameters

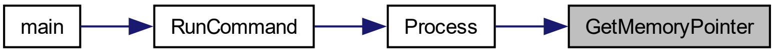
shared_mem_id	The id of the shared memory

Returns

A pointer to the shared memory, or -1 if it fails.

Definition at line 26 of file memShare.c.

Here is the caller graph for this function:

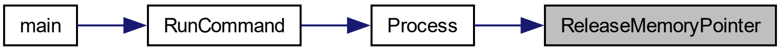


5.19.1.4 ReleaseMemoryPointer()

```
int ReleaseMemoryPointer (
            void * shmaddr )
```

Definition at line 31 of file memShare.c.

Here is the caller graph for this function:



5.20 memShare.c

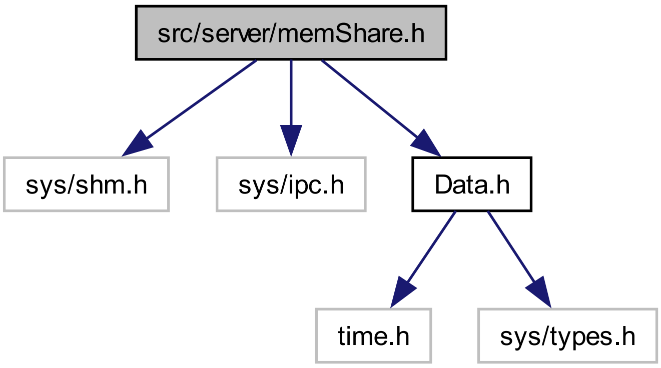
Go to the documentation of this file.

```
00001
00004 #include "memShare.h"
00005 #include <string.h>
00006 #include <stdio.h>
00007
00008 /*
00009 Todo:  Error handling and printing
00010 */
00011
00012 int CreateSharedMemory()
00013 {
    return shmget(MEM_KEY, MEM_SIZE, IPC_CREAT | MEM_PERMISSIONS);
}

int DestroySharedMemory()
{
    int shm_id = shmget(MEM_KEY, MEM_SIZE, 0);
    int control_result = shmctl(shm_id, IPC_RMID, 0);
    if (control_result != -1)
        return 0;
    return control_result;
}

void *GetMemoryPointer(int shared_mem_id)
{
    return shmat(shared_mem_id, NULL, 0);
}

int ReleaseMemoryPointer(void *shmaddr)
{
    return shmdt(shmaddr);
}
```

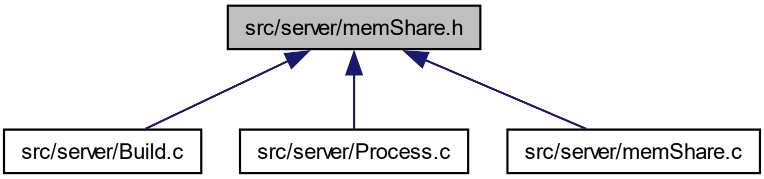
5.21 src/server/memShare.h File Reference

```
#include <sys/shm.h>
#include <sys/ipc.h>
#include "Data.h"
```

Include dependency graph for memShare.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define MEM_KEY 0x727

  Declarations for functions that operate on a shared memory segment.
- #define MEM_PERMISSIONS 0664
- #define MEM_SIZE DATA_SIZE ∗DATA_NUM_RECORDS

Functions

- int CreateSharedMemory ()

  Definitions for functions that operate on a shared memory segment.
- int DestroySharedMemory ()
- void ∗ GetMemoryPointer (int shared_mem_id)
- int ReleaseMemoryPointer (void ∗shmaddr)

5.21.1 Macro Definition Documentation

5.21.1.1 MEM_KEY

```
#define MEM_KEY 0x727
```

Declarations for functions that operate on a shared memory segment.

Definition at line 14 of file memShare.h.

5.21.1.2 MEM_PERMISSIONS

```
#define MEM_PERMISSIONS 0664
```

Definition at line 25 of file memShare.h.

5.21.1.3 MEM_SIZE

```
#define MEM_SIZE DATA_SIZE *DATA_NUM_RECORDS
```

Definition at line 30 of file memShare.h.

5.21.2 Function Documentation

5.21.2.1 CreateSharedMemory()

```
int CreateSharedMemory ( )
```

Definitions for functions that operate on a shared memory segment.

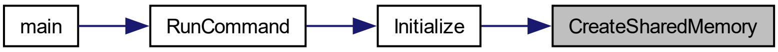
CreateSharedMemory retrieves a shared memory ID that can be used to access or delete shared memory.

Returns

A shared memory ID that can be used with other 'shm' commands to access shared memory.

Definition at line 12 of file memShare.c.

Here is the caller graph for this function:



5.21.2.2 DestroySharedMemory()

```
int DestroySharedMemory ( )
```

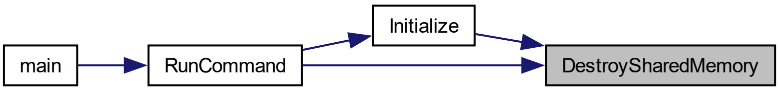
Flags the shared memory segment for deallocation. Returns the result of that operation.

Returns

0 if succesful. 1 if not succesful. Errno will be set.

Definition at line 17 of file memShare.c.

Here is the caller graph for this function:



5.21.2.3 GetMemoryPointer()

```
void * GetMemoryPointer (
            int shared_mem_id )
```

"Attaches" to the shared memory, returning a memory pointer to the shared memory.

Calls 'shmat(shared_mem_id, NULL, 0)`;

Parameters

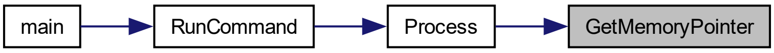
shared_mem_id	The id of the shared memory

Returns

A pointer to the shared memory, or -1 if it fails.

Definition at line 26 of file memShare.c.

Here is the caller graph for this function:

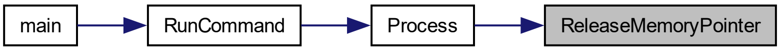


5.21.2.4 ReleaseMemoryPointer()

```
int ReleaseMemoryPointer (
              void * shmaddr )
```

Definition at line 31 of file memShare.c.

Here is the caller graph for this function:



5.22 memShare.h

Go to the documentation of this file.

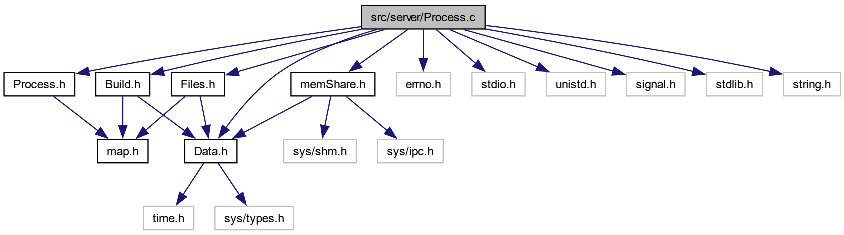
```
#ifndef MEM_SHARE_H
#define MEM_SHARE_H
#include <sys/shm.h>
#include <sys/ipc.h>
#include "Data.h"

/*
The shared memory key that clients and servers will use to identify the segment.
*/
#define MEM_KEY 0x727

```

```
/*
Memory permissions are:
Self:      RW    110 = 6
Group:     R     100 = 4
Others:  R     100 = 4
- All groups can read.
- Self can write.
- None can execute.
*/
#define MEM_PERMISSIONS 0664

/*
The memory allocation must as large as the data size times the number of records.
*/
#define MEM_SIZE DATA_SIZE *DATA_NUM_RECORDS

int CreateSharedMemory();

int DestroySharedMemory();

void *GetMemoryPointer(int shared_mem_id);

/*
Release a shm memory pointer.
*/
int ReleaseMemoryPointer(void *shmaddr);

#endif
```

5.23 src/server/Process.c File Reference

```
#include "Process.h"
#include "Files.h"
#include "Data.h"
#include "Build.h"
#include "memShare.h"
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <string.h>
```

Include dependency graph for Process.c:



Functions

- short DoesLockfileExist ()
- int CreateLockfile ()
- int DeleteLockfile ()
- int TerminateExistingServer ()
- int IndicateRereadNeeded ()

- int IndicateRereadDone ()
- short IsRereadNeeded ()
- void SignalHandle (int signo)
- int Initialize ()
- void Process (int shm_id)
- void HelpCommand ()
- void RunCommand ()
- void StopCommand ()
- void ResetCommand ()
- void RunHeadless (char ∗processName)

Variables

- map ∗ student_map

  Definitions for functions that manage control flow.
- map ∗ initial_cumulative_times
- short is_stopping = 0

5.23.1 Function Documentation

5.23.1.1 CreateLockfile()

```
int CreateLockfile ( )
```

Creates a lockfile.

Warning

> This should only be called by a running server process when a lockfile does not already exist.

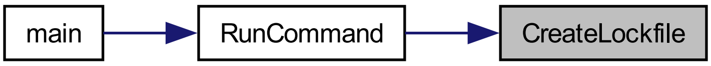
The lockfile will carry a 'data reset' signal and a process ID. CreateLockfile will write the current processes PID.

Returns

> -1 if fopen failed, otherwise 0.

Definition at line 27 of file Process.c.

Here is the caller graph for this function:



5.23.1.2 DeleteLockfile()

```
int DeleteLockfile ( )
```

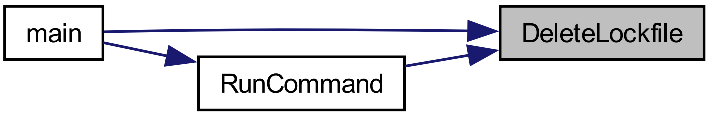
Deletes the lockfile.

Returns

0 on success, -1 on failure.

Definition at line 39 of file Process.c.

Here is the caller graph for this function:



5.23.1.3 DoesLockfileExist()

```
short DoesLockfileExist ( )
```

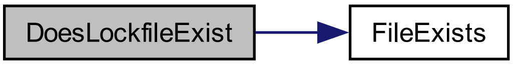
Determines if lockfile exists, which indicates that a server process is already running.

Returns

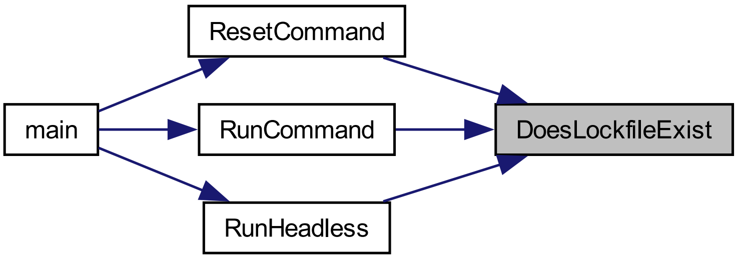
0 if lockfile does not exist, 1 if it does.

Definition at line 22 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.23.1.4 HelpCommand()

```
void HelpCommand ( )
```

Displays the commands available to the user.

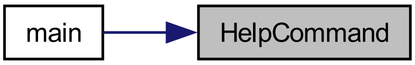
Note

To execute the command, pass "help" as an argument to the program.

This command will also run if arg num is incorrect or if invalid option is entered.

Definition at line 214 of file Process.c.

Here is the caller graph for this function:



5.23.1.5 IndicateRereadDone()

```
int IndicateRereadDone ( )
```

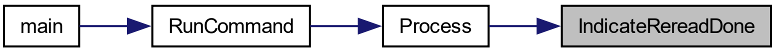
If we re-read the users file, we can indicate that we have done so by setting the re-read flag back to 0.

Warning

should only be called by main process.

Definition at line 79 of file Process.c.

Here is the caller graph for this function:



5.23.1.6 IndicateRereadNeeded()

```
int IndicateRereadNeeded ( )
```

If we reset the user data, we need to indicate to the running process that a re-read is needed. This changes the flag in the lockfile to 1, but keeps the same process ID as before there.

Warning

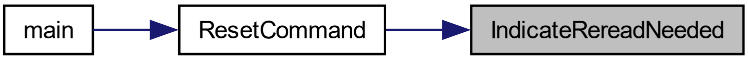
should only be called by non main processes

Returns

-1 if lockfile not found., 0 if success, or an error number if some other error

Definition at line 62 of file Process.c.

Here is the caller graph for this function:



5.23.1.7 Initialize()

```
int Initialize ( )
```

Run once at program start. Calls functions from other modules to do the following.

Note

(1) - Create an initial user data file if it doesn't exist

(2) - Initialize the students array.

(3) - Initialize the students map.

(4) - Read the data from the user data file into the map/array.

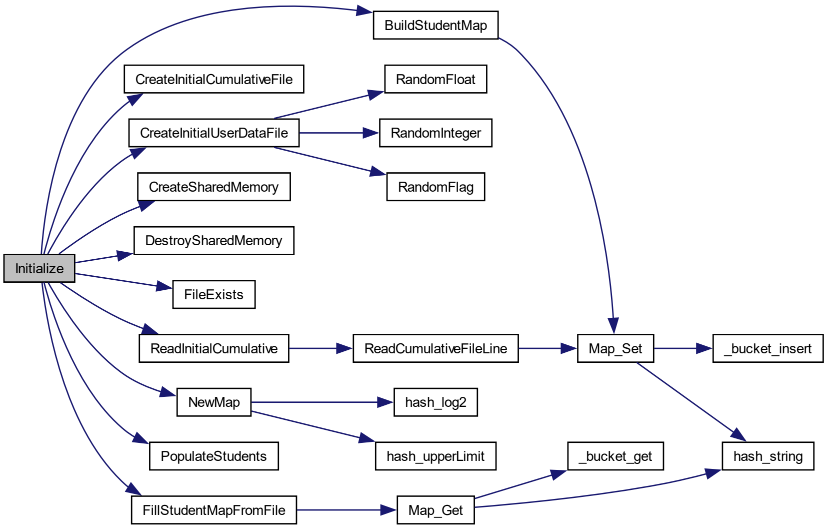
(5) - Initializes the shared memory segment.

Returns

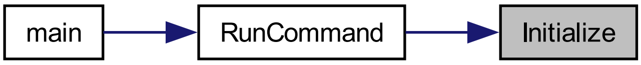
The ID of the shared memory segment. If -1, there was an error.

Definition at line 117 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.23.1.8 IsRereadNeeded()

```
short IsRereadNeeded ( )
```

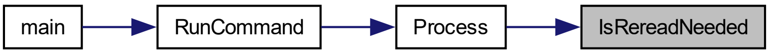
Reads the lockfile for the reread flag.

Warning

> Lockfile should exist - should be called by the server in the main process loop

Definition at line 96 of file Process.c.

Here is the caller graph for this function:



5.23.1.9 Process()

```
void Process (
              int shm_id )
```

Called repeatedly with a delay.

Note

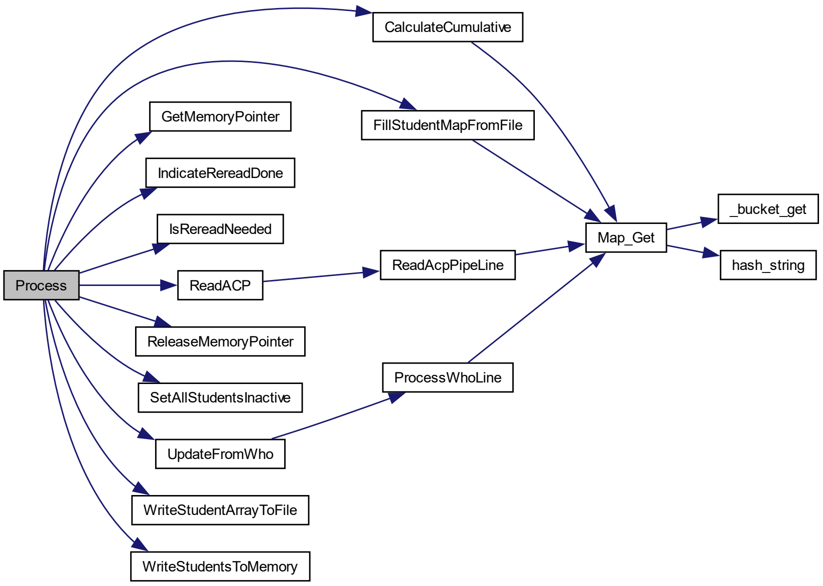
> (1) - Sets all users to inactive.
>
> (2) - Reads the result of the `who` command, setting some users to active, and possibly changing 'dirty' and last login times.
>
> (3) - Overwrites the user data file if we are dirty.
>
> (4) - Sets dirty to false.
>
> (5) - Re-writes the shared memory.

Parameters

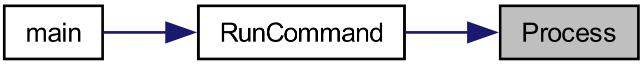
shm←_id	The ID of the shared memory segment.

Definition at line 167 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.23.1.10 ResetCommand()

```
void ResetCommand ( )
```

Deletes and re-creates the static-user-data file and cumulative login file.

Note

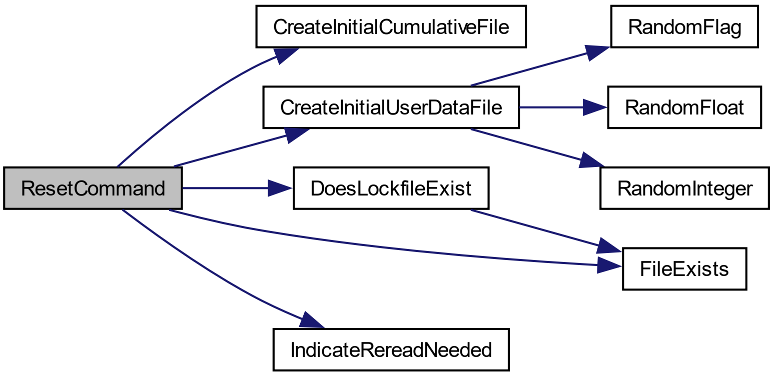
To execute the command, pass "reset" as an argument to the program.

Warning

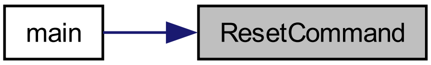
This will clear login times.

Definition at line 288 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.23.1.11 RunCommand()

```
void RunCommand ( )
```

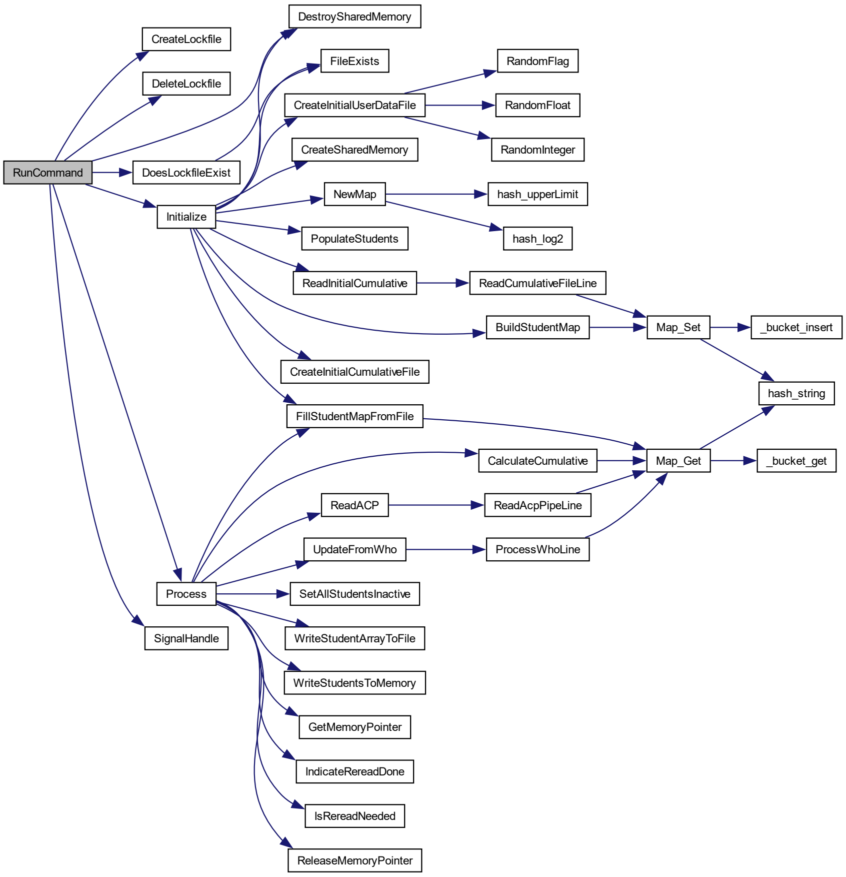
If a server exists, stops it. Begins the process loop.

Note

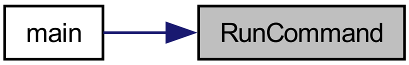
To execute the command, pass "run" as an argument to the program.

Definition at line 225 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.23.1.12 RunHeadless()

```
void RunHeadless (
             char * processName )
```

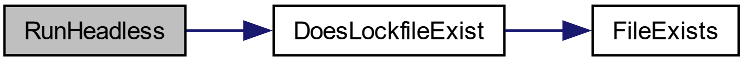
Uses nohup ./{processName} run to run the prodess headlessly.

Parameters

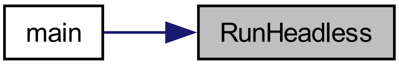
processName	The name of the currently running process.

Definition at line 327 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.23.1.13 SignalHandle()

```
void SignalHandle (
             int signo )
```

Called by a new server process, telling this server process to shut down. This sets 'is_stopping' to true, which shuts down the server gracefully, writing any necessary data to the user data file, then deleting the lockfile.

Parameters

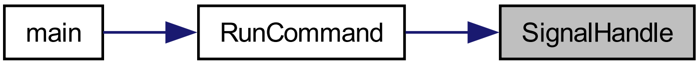
signo	The signal number. Will be SIGTERM from the other server process or SIGINT if interrupted from the console.

Returns

0 on success, -1 on error

Definition at line 104 of file Process.c.

Here is the caller graph for this function:



5.23.1.14 StopCommand()

```
void StopCommand ( )
```

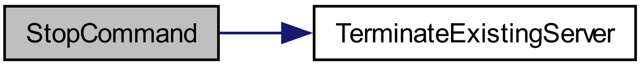
Stops an existing server process if it is running.

Note

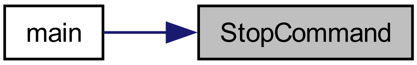
To execute the command, pass "stop" as an argument to the program.

Definition at line 263 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.23.1.15 TerminateExistingServer()

```
int TerminateExistingServer ( )
```

Reads the lockfile to get the ID of the process that created it.

Sends a SIGTERM signal to that process.

Warning

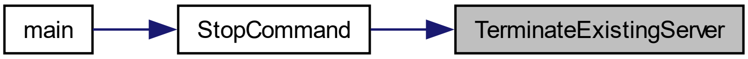
lockfile should be confirmed to exist

Returns

-1 if file doesn't exist, -2, if no valid process ID existed in the file, 1 if sending the kill signal failed.

Definition at line 44 of file Process.c.

Here is the caller graph for this function:



5.23.2 Variable Documentation

5.23.2.1 initial_cumulative_times

```
map* initial_cumulative_times
```

Definition at line 18 of file Process.c.

5.23.2.2 is_stopping

```
short is_stopping = 0
```

If 0, the server is running and looping, rereading and writing every second. If 1, it is stopping and shutting down.

Definition at line 113 of file Process.c.

5.23.2.3 student_map

```
map* student_map
```

Definitions for functions that manage control flow.

Definition at line 17 of file Process.c.

5.24 Process.c

Go to the documentation of this file.

```
00001
00005 #include "Process.h"
00006 #include "Files.h"
00007 #include "Data.h"
00008 #include "Build.h"
00009 #include "memShare.h"
00010 #include <errno.h>
00011 #include <stdio.h>
00012 #include <unistd.h>
00013 #include <signal.h>
00014 #include <stdlib.h>
00015 #include <string.h>
00016
00017 map *student_map;
00018 map *initial_cumulative_times;
00019
00020 // ~~~~~~~~~~~~~~~ Lockfile Commands ~~~~~~~~~~~~~~~~~~~~~
00021
00022 short DoesLockfileExist()
00023 {
00024     return FileExists(LOCKFILE);
00025 }
00026
00027 int CreateLockfile()
00028 {
00029     FILE *file = fopen(LOCKFILE, "w");
00030     if (file == NULL)
00031     {
00032         return -1;
00033     }
00034     fprintf(file, "0 %d", getpid());
00035     fclose(file);
00036     return 0;
00037 }
00038
00039 int DeleteLockfile()
00040 {
00041     return remove(LOCKFILE);
00042 }
00043
00044 int TerminateExistingServer()
00045 {
00046     FILE *file = fopen(LOCKFILE, "r");
00047     if (file == NULL)
00048     {
00049         return -1;
00050     }
00051     int need_rewrite;
00052     int pid = 0;
00053     fscanf(file, "%d %d", &need_rewrite, &pid);
00054     fclose(file);
00055     if (pid > 0)
00056     {
00057         return kill(pid, SIGTERM);
00058     }
00059     return -2;
00060 }
00061
00062 int IndicateRereadNeeded()
00063 {
00064     FILE *file = fopen(LOCKFILE, "r+");
00065     if (file == NULL)
00066     {
00067         return -1;
00068     }
00069     int err = 0;
```

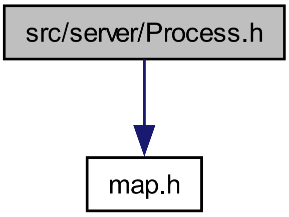
```
    err = fseek(file, 0, SEEK_SET);
    if (!err)
    {
        fputc(’1’, file);
    }
    err = fclose(file);
    return err;
}

int IndicateRereadDone()
{
    FILE *file = fopen(LOCKFILE, "r+");
    if (file == NULL)
    {
        return -1;
    }
    int err = 0;
    err = fseek(file, 0, SEEK_SET);
    if (!err)
    {
        fputc(’0’, file);
    }
    err = fclose(file);
    return err;
}

short IsRereadNeeded()
{
    FILE *file = fopen(LOCKFILE, "r");
    char firstc = fgetc(file);
    fclose(file);
    return firstc == ’1’;
}

void SignalHandle(int signo)
{
    printf("Received shutdown signal.\n");
    if (signo == SIGINT || signo == SIGTERM)
    {
        is_stopping = 1;
    }
}

short is_stopping = 0;

// ~~~~~~~~~~~~~~~ CLI Commands ~~~~~~~~~~~~~~~~~~~~~~~~~~

int Initialize()
{
    int err;
    if (!FileExists(STATIC_USER_DATA_FILE))
    {
        printf("%s does not exist.  Creating.\n", STATIC_USER_DATA_FILE);
        err = CreateInitialUserDataFile(STATIC_USER_DATA_FILE, Data_IDs, DATA_NUM_RECORDS);
        if (err)
        {
            printf("Problem creating %s!\n", STATIC_USER_DATA_FILE);
        }
    }
    if (!FileExists(STATIC_USER_CUMULATIVE_FILE))
    {
        printf("%s does not exist.  Creating.\n", STATIC_USER_CUMULATIVE_FILE);
        err = CreateInitialCumulativeFile(STATIC_USER_CUMULATIVE_FILE);
        if (err)
        {
            printf("Problem creating %s!\n", STATIC_USER_CUMULATIVE_FILE);
        }
    }
    PopulateStudents(Data_IDs, Data_Names, DATA_NUM_RECORDS);
    student_map = NewMap(50);
    BuildStudentMap(student_map, students, DATA_NUM_RECORDS);
    err = FillStudentMapFromFile(student_map, STATIC_USER_DATA_FILE, Data_IDs, DATA_NUM_RECORDS);
    if (err)
    {
        printf("Problem filling student map from %s!\n", STATIC_USER_DATA_FILE);
    }
    printf("Student data retrieved from file.\n");

    initial_cumulative_times = NewMap(50);
    err = ReadInitialCumulative(initial_cumulative_times, STATIC_USER_CUMULATIVE_FILE);
    if (err)
    {
        printf("Failed to read %s.  Cumulative times may be wrong!", STATIC_USER_CUMULATIVE_FILE);
    }

    dirty = 0;

```

```
    int shmid = CreateSharedMemory();
    if (shmid == -1)
    {
        DestroySharedMemory();
        shmid = CreateSharedMemory();
    }
    printf("Shared memory allocated.\n");
    return shmid;
}

void Process(int shm_id)
{
    if (IsRereadNeeded())
    {
        printf("\nReread indicated - rechecking user data file.");
        FillStudentMapFromFile(student_map, STATIC_USER_DATA_FILE, Data_IDs, DATA_NUM_RECORDS);
        IndicateRereadDone();
    }
    SetAllStudentsInactive(students, DATA_NUM_RECORDS);
    int err = ReadACP(student_map);
    if (err)
    {
        printf("Error piping ac -p command!  \n");
    }
    else
    {
        CalculateCumulative(students, DATA_NUM_RECORDS, initial_cumulative_times);
    }
    err = UpdateFromWho(student_map);
    if (err)
    {
        perror("Error updating from who!");
    }
    if (dirty)
    {
        err = WriteStudentArrayToFile(students, DATA_NUM_RECORDS, STATIC_USER_DATA_FILE);
        if (err)
        {
            printf("\nError updating %s!", STATIC_USER_DATA_FILE);
        }
        else
        {
            dirty = 0;
        }
    }
    void *ptr = GetMemoryPointer(shm_id);
    if (ptr == (void *)-1)
    {
        perror("Error attaching to shared memory");
    }
    else
    {
        WriteStudentsToMemory(ptr, students, DATA_NUM_RECORDS);
        ReleaseMemoryPointer(ptr);
    }
}

void HelpCommand()
{
    printf("\nUsage:  server [OPTION]\n\n");
    printf("Options:  \n");
    printf("\thelp\t\t\tShows the possible program commands\n");
    printf("\treset\t\t\tRegenerates the user data file\n");
    printf("\tstop\t\t\tStops an existing server process if it is running\n");
    printf("\trun\t\t\tCreates a new server with output to the shell if a server isn’t already running.\n");
    printf("\theadless\t\tCreates a new headless server if a server isn’t already running.\n\n");
}

void RunCommand()
{
    printf("\nRunning server.\n");
    if (DoesLockfileExist())
    {
        printf("\nServer is already running.  Run ’server stop’ to shut it down first.\n");
        return;
    }
    int err = CreateLockfile();
    if (err)
    {
        printf("\nFailed to create lockfile!  Exiting.\n");
        return;
    }
    int shm_id = Initialize();
    signal(SIGTERM, SignalHandle);
    signal(SIGINT, SignalHandle);
    printf("Server started.\n");
    fflush(stdout);
    while (!is_stopping)
    {
        Process(shm_id);
        sleep(1);
    }
    printf("Server shutting down.\n");
    err = DeleteLockfile();
    if (err)
    {
        printf("Failed to delete lockfile!\n");
    }
    err = DestroySharedMemory();
    if (err)
    {
        printf("Failed to destroy shared memory!\n");
    }
    printf("Server terminated.\n");
}

void StopCommand()
{
    printf("\nStopping server...\n");
    int err = TerminateExistingServer();
    if (err)
    {
        if (err == -1)
        {
            printf("Server isn’t running.\n");
        }
        else if (err == -2)
        {
            printf("Lockfile did not contain a valid process id!\n");
        }
        else
        {
            printf("Sending terminate signal failed!\n");
        }
    }
    else
    {
        printf("Server terminated.\n");
    }
}

void ResetCommand()
{
    int err;

    if (FileExists(STATIC_USER_DATA_FILE))
    {
        printf("User data file exists.  Deleting...\n");
        remove(STATIC_USER_DATA_FILE);
    }

    printf("Creating new data file.\n");
    err = CreateInitialUserDataFile(STATIC_USER_DATA_FILE, Data_IDs, DATA_NUM_RECORDS);
    if (err)
    {
        printf("Problem creating %s!\n", STATIC_USER_DATA_FILE);
    }
    else
    {
        printf("%s created.\n", STATIC_USER_DATA_FILE);
    }

    printf("Creating new cumulative file.\n");
    err = CreateInitialCumulativeFile(STATIC_USER_CUMULATIVE_FILE);
    if (err)
    {
        printf("Problem creating %s!\n", STATIC_USER_CUMULATIVE_FILE);
    }
    else
    {
        printf("%s created.\n", STATIC_USER_CUMULATIVE_FILE);
    }

    if (DoesLockfileExist())
    {
        printf("Indicated re-read to running server process.\n");
        IndicateRereadNeeded();
    }
}

void RunHeadless(char *processName)
{
    if (DoesLockfileExist())
    {
        printf("Server process already running.\n");
        return;
    }
    char commandFront[] = " nohup ";
    char commandEnd[] = " run & exit";
    size_t comm_length = strlen(commandFront) + strlen(commandEnd) + strlen(processName) + 1;
    char *commandFull = malloc(comm_length * sizeof(char));
    memset(commandFull, 0, comm_length * sizeof(char));
    strcpy(commandFull, commandFront);
    strcat(commandFull, processName);
    strcat(commandFull, commandEnd);

    printf("Executing:  %s\n", commandFull);
    popen(commandFull, "we");
    printf("Server running headlessly.\n");
}
```

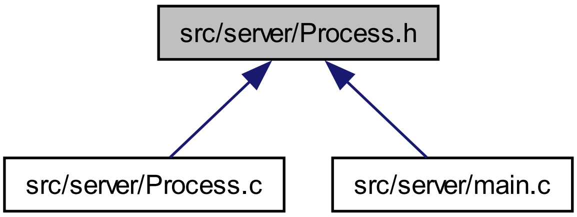
5.25 src/server/Process.h File Reference

```
#include "map.h"
```

Include dependency graph for Process.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define LOCKFILE "/tmp/ecet-server.lock"

  Definitions for functions that manage control flow. This module handles the processes that this server might execute. It calls functions from the other modules to realize program changes.

Functions

- short DoesLockfileExist ()
- int CreateLockfile ()
- int DeleteLockfile ()
- int TerminateExistingServer ()
- int IndicateRereadNeeded ()
- int IndicateRereadDone ()
- short IsRereadNeeded ()
- void SignalHandle (int signo)
- int Initialize ()
- void Process (int shm_id)
- void ClearCommand ()
- void ResetCommand ()
- void StopCommand ()
- void RunCommand ()
- void HelpCommand ()
- void RunHeadless (char ∗processName)

Variables

- short is_stopping
- map ∗ student_map

  Definitions for functions that manage control flow.

5.25.1 Macro Definition Documentation

5.25.1.1 LOCKFILE

```
#define LOCKFILE "/tmp/ecet-server.lock"
```

Definitions for functions that manage control flow. This module handles the processes that this server might execute. It calls functions from the other modules to realize program changes.

The lockfile serves as a signal to subsequent processes as to whether or not server is already running.

It carries two pieces of data. The first is a 0 or 1, telling the server whether user data has been reset and needs to be re-read.

The second is an int corresponding to the process ID, so a close signal can be sent to the running process when a user enters `server close`.

Definition at line 17 of file Process.h.

5.25.2 Function Documentation

5.25.2.1 ClearCommand()

```
void ClearCommand ( )
```

Clears / Deallocates the shared virtual memory segment.

Note

To execute the command, pass "clear" as an argument to the program.

5.25.2.2 CreateLockfile()

```
int CreateLockfile ( )
```

Creates a lockfile.

Warning

This should only be called by a running server process when a lockfile does not already exist.

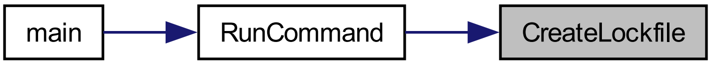
The lockfile will carry a 'data reset' signal and a process ID. CreateLockfile will write the current processes PID.

Returns

-1 if fopen failed, otherwise 0.

Definition at line 27 of file Process.c.

Here is the caller graph for this function:



5.25.2.3 DeleteLockfile()

```
int DeleteLockfile ( )
```

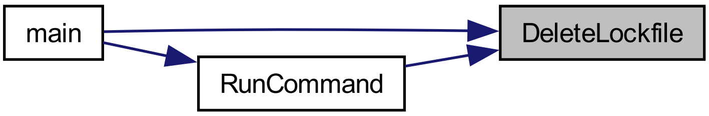
Deletes the lockfile.

Returns

0 on success, -1 on failure.

Definition at line 39 of file Process.c.

Here is the caller graph for this function:



5.25.2.4 DoesLockfileExist()

```
short DoesLockfileExist ( )
```

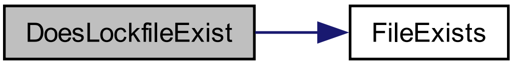
Determines if lockfile exists, which indicates that a server process is already running.

Returns

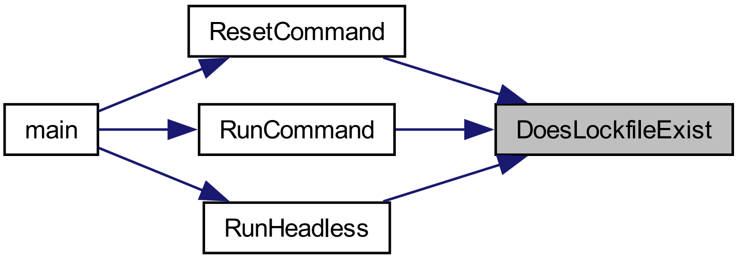
0 if lockfile does not exist, 1 if it does.

Definition at line 22 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.25.2.5 HelpCommand()

```
void HelpCommand ( )
```

Displays the commands available to the user.

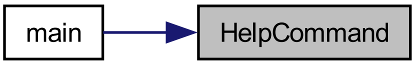
Note

To execute the command, pass "help" as an argument to the program.

This command will also run if arg num is incorrect or if invalid option is entered.

Definition at line 214 of file Process.c.

Here is the caller graph for this function:



5.25.2.6 IndicateRereadDone()

```
int IndicateRereadDone ( )
```

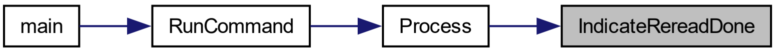
If we re-read the users file, we can indicate that we have done so by setting the re-read flag back to 0.

Warning

should only be called by main process.

Definition at line 79 of file Process.c.

Here is the caller graph for this function:



5.25.2.7 IndicateRereadNeeded()

```
int IndicateRereadNeeded ( )
```

If we reset the user data, we need to indicate to the running process that a re-read is needed. This changes the flag in the lockfile to 1, but keeps the same process ID as before there.

Warning

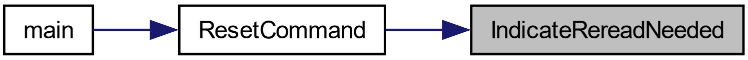
should only be called by non main processes

Returns

-1 if lockfile not found., 0 if success, or an error number if some other error

Definition at line 62 of file Process.c.

Here is the caller graph for this function:



5.25.2.8 Initialize()

```
int Initialize ( )
```

Run once at program start. Calls functions from other modules to do the following.

Note

(1) - Create an initial user data file if it doesn't exist

(2) - Initialize the students array.

(3) - Initialize the students map.

(4) - Read the data from the user data file into the map/array.

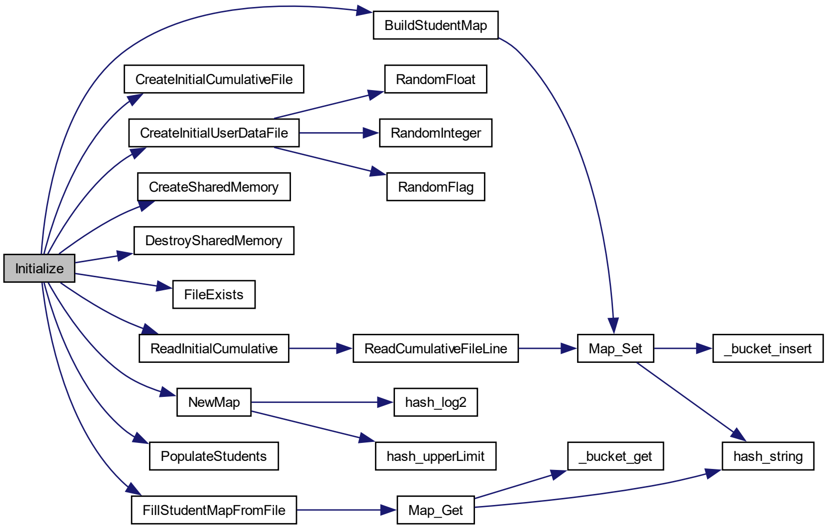
(5) - Initializes the shared memory segment.

Returns

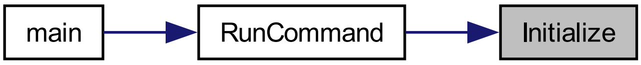
The ID of the shared memory segment. If -1, there was an error.

Definition at line 117 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.25.2.9 IsRereadNeeded()

```
short IsRereadNeeded ( )
```

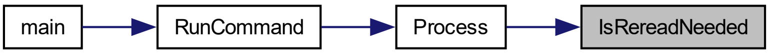
Reads the lockfile for the reread flag.

Warning

Lockfile should exist - should be called by the server in the main process loop

Definition at line 96 of file Process.c.

Here is the caller graph for this function:



5.25.2.10 Process()

```
void Process (
             int shm_id )
```

Called repeatedly with a delay.

Note

(1) - Sets all users to inactive.

(2) - Reads the result of the `who` command, setting some users to active, and possibly changing 'dirty' and last login times.

(3) - Overwrites the user data file if we are dirty.

(4) - Sets dirty to false.

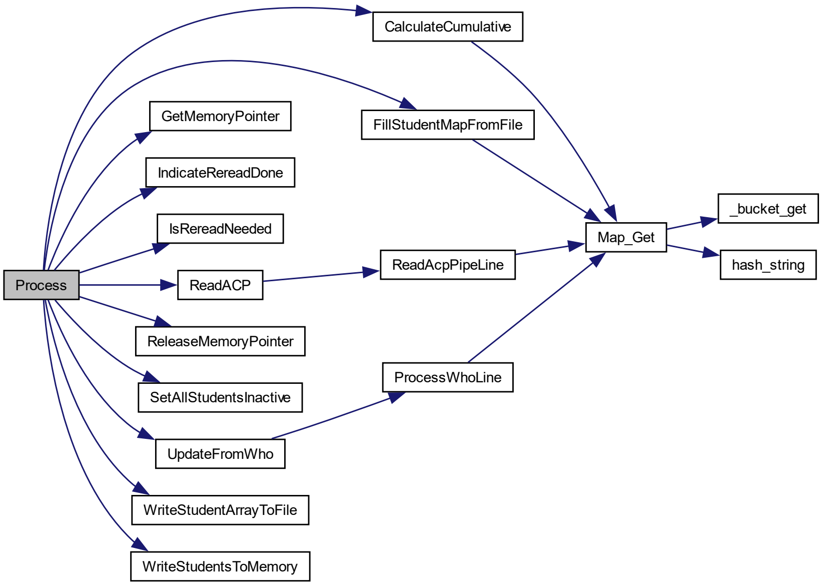
(5) - Re-writes the shared memory.

Parameters

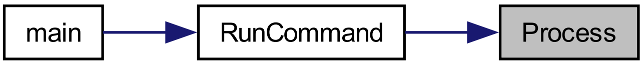
shm_id	The ID of the shared memory segment.

Definition at line 167 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.25.2.11 ResetCommand()

```
void ResetCommand ( )
```

Deletes and re-creates the static-user-data file and cumulative login file.

Note

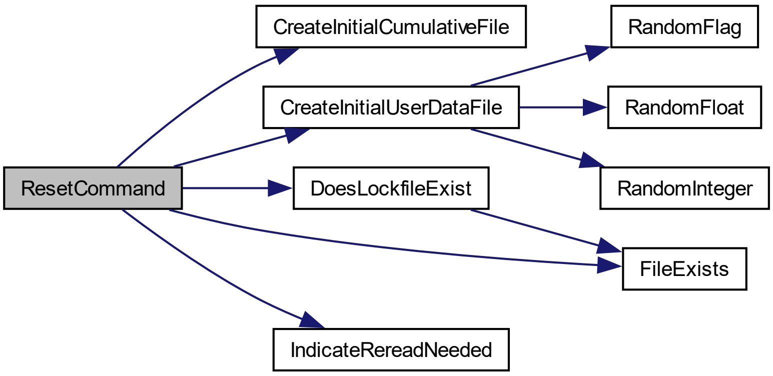
To execute the command, pass "reset" as an argument to the program.

Warning

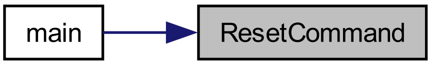
This will clear login times.

Definition at line 288 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.25.2.12 RunCommand()

```
void RunCommand ( )
```

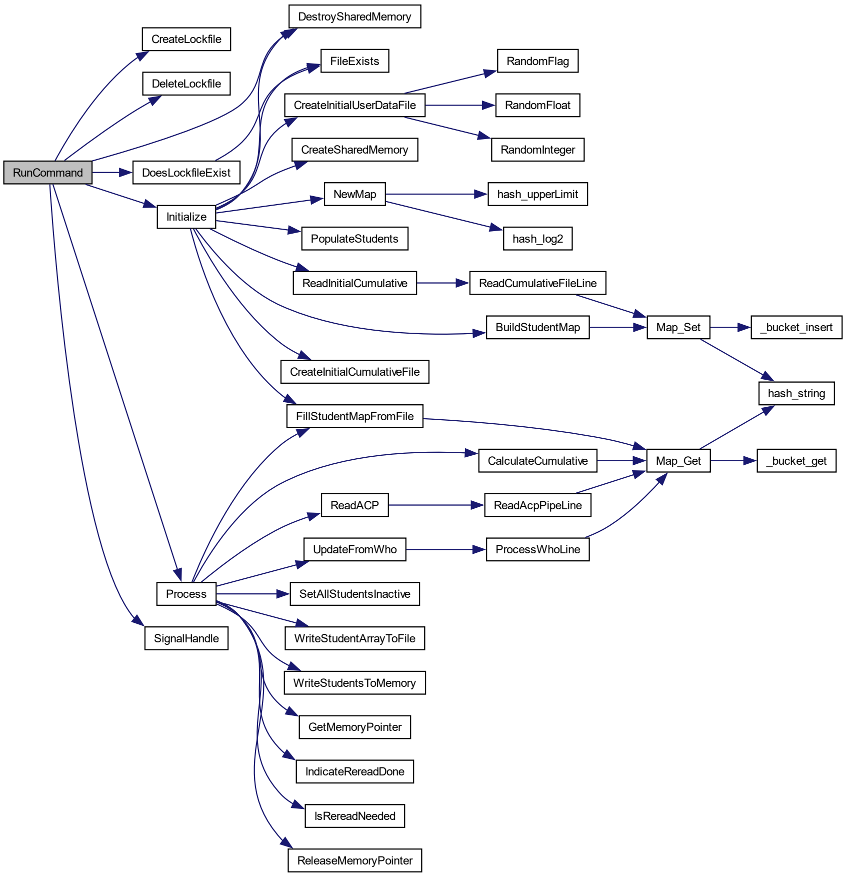
If a server exists, stops it. Begins the process loop.

Note

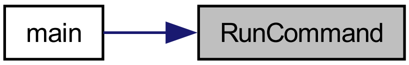
To execute the command, pass "run" as an argument to the program.

Definition at line 225 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.25.2.13 RunHeadless()

```
void RunHeadless (
             char * processName )
```

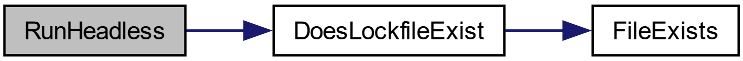
Uses nohup ./{processName} run to run the prodess headlessly.

Parameters

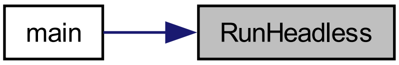
processName	The name of the currently running process.

Definition at line 327 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.25.2.14 SignalHandle()

```
void SignalHandle (
             int signo )
```

Called by a new server process, telling this server process to shut down. This sets 'is_stopping' to true, which shuts down the server gracefully, writing any necessary data to the user data file, then deleting the lockfile.

Parameters

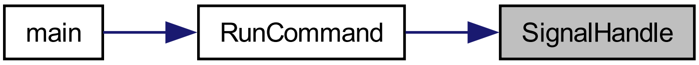
signo	The signal number. Will be SIGTERM from the other server process or SIGINT if interrupted from the console.

Returns

0 on success, -1 on error

Definition at line 104 of file Process.c.

Here is the caller graph for this function:



5.25.2.15 StopCommand()

```
void StopCommand ( )
```

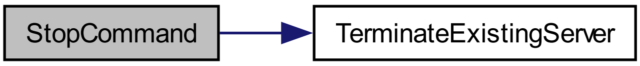
Stops an existing server process if it is running.

Note

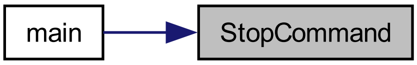
To execute the command, pass "stop" as an argument to the program.

Definition at line 263 of file Process.c.

Here is the call graph for this function:



Here is the caller graph for this function:



5.25.2.16 TerminateExistingServer()

```
int TerminateExistingServer ( )
```

Reads the lockfile to get the ID of the process that created it.

Sends a SIGTERM signal to that process.

Warning

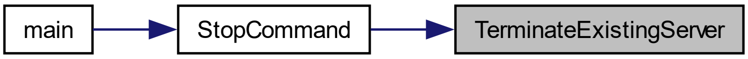
lockfile should be confirmed to exist

Returns

-1 if file doesn't exist, -2, if no valid process ID existed in the file, 1 if sending the kill signal failed.

Definition at line 44 of file Process.c.

Here is the caller graph for this function:



5.25.3 Variable Documentation

5.25.3.1 is_stopping

```
short is_stopping [extern]
```

If 0, the server is running and looping, rereading and writing every second. If 1, it is stopping and shutting down.

Definition at line 113 of file Process.c.

5.25.3.2 student_map

```
map* student_map [extern]
```

Definitions for functions that manage control flow.

Definition at line 17 of file Process.c.

5.26 Process.h

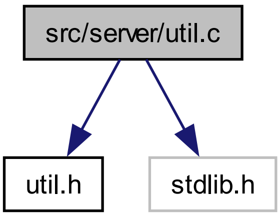
Go to the documentation of this file.

```
#ifndef Process_h
#define Process_h
#include "map.h"

#define LOCKFILE "/tmp/ecet-server.lock"

// ~~~~~~~~~~~~~~~ Lockfile Commands ~~~~~~~~~~~~~~~~~~~~~

short DoesLockfileExist();

int CreateLockfile();

int DeleteLockfile();

int TerminateExistingServer();

int IndicateRereadNeeded();

int IndicateRereadDone();

short IsRereadNeeded();

void SignalHandle(int signo);

extern short is_stopping;

// ~~~~~~~~~~~~~~~ CLI Commands ~~~~~~~~~~~~~~~~~~~~~~~~~~

extern map *student_map;

int Initialize();

void Process(int shm_id);

void ClearCommand();

void ResetCommand();

void StopCommand();

void RunCommand();

void HelpCommand();

void RunHeadless(char *processName);

#endif
```

5.27 src/server/util.c File Reference

```
#include "util.h"
#include <stdlib.h>
```

Include dependency graph for util.c:



Functions

- int RandomInteger (int min, int max)

  Definitions for helper functions.
- float RandomFloat (float min, float max)
- short RandomFlag (float percentage_chance)

5.27.1 Function Documentation

5.27.1.1 RandomFlag()

```
short RandomFlag (
             float percentage_chance )
```

Returns 1, percentage_chance of the time.

Parameters

percentage_chance	The chance to return 1.

Note

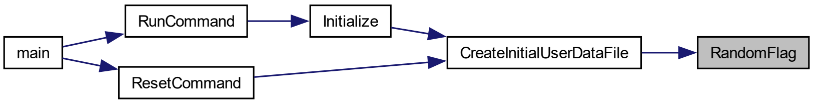
If percentage_chance $>$ 1, this will always return true.

Returns

1 or 0

Definition at line 21 of file util.c.

Here is the caller graph for this function:



5.27.1.2 RandomFloat()

```
float RandomFloat (
             float min,
             float max )
```

Returns a float between min and max.

Parameters

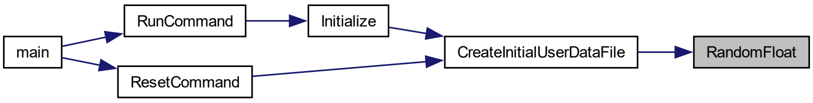
min	The minimum, inclusive.
max	The maximum, inclusive.

Returns

A random integer between min and max.

Definition at line 14 of file util.c.

Here is the caller graph for this function:



5.27.1.3 RandomInteger()

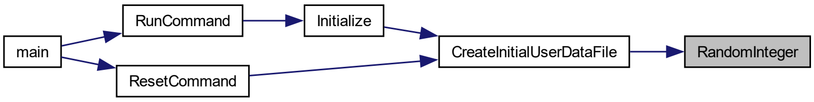
```
int RandomInteger (
             int min,
             int max )
```

Definitions for helper functions.

Declarations for helper functions.

Definition at line 8 of file util.c.

Here is the caller graph for this function:



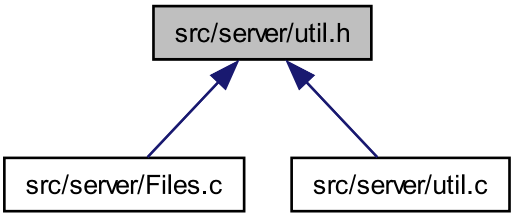
5.28 util.c

Go to the documentation of this file.

```
#include "util.h"

#include <stdlib.h>

int RandomInteger(int min, int max)
{
    int r_add = rand() % (max - min + 1);
    return r_add + min;
}

float RandomFloat(float min, float max)
{
    float dif = max - min;
    int rand_int = rand() % (int)(dif * 10000);
    return min + (float)rand_int / 10000.0;
}

short RandomFlag(float percentage_chance)
{
    float random_value = (float)rand() / RAND_MAX;
    if (random_value < percentage_chance)
    {
        return 1;
    }
    return 0;
}
```

5.29 src/server/util.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- int RandomInteger (int min, int max)

    Declarations for helper functions.
- float RandomFloat (float min, float max)
- short RandomFlag (float percentage_chance)

5.29.1 Function Documentation

5.29.1.1 RandomFlag()

```
short RandomFlag (
             float percentage_chance )
```

Returns 1, percentage_chance of the time.

Parameters

percentage_chance	The chance to return 1.

Note

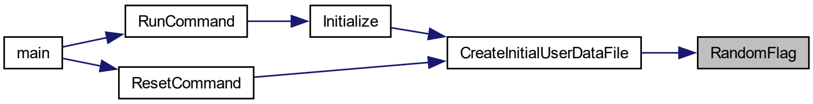
If percentage_chance $>$ 1, this will always return true.

Returns

1 or 0

Definition at line 21 of file util.c.

Here is the caller graph for this function:



5.29.1.2 RandomFloat()

```
float RandomFloat (
             float min,
             float max )
```

Returns a float between min and max.

Parameters

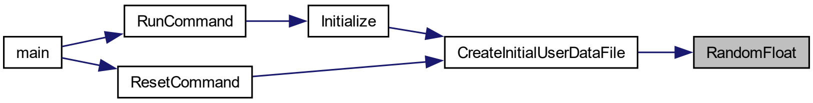
min	The minimum, inclusive.
max	The maximum, inclusive.

Returns

A random integer between min and max.

Definition at line 14 of file util.c.

Here is the caller graph for this function:



5.29.1.3 RandomInteger()

```
int RandomInteger (
              int min,
              int max )
```

Declarations for helper functions.

Contains utility functions that are not coupled to any other data or structures in the program. Returns an integer between min and max.

Parameters

min	The minimum, inclusive.
max	The maximum, inclusive.

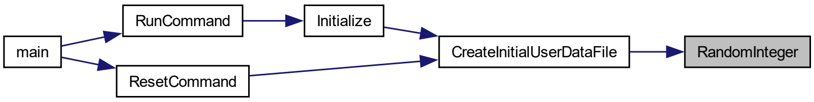
Returns

A random integer between min and max.

Declarations for helper functions.

Definition at line 8 of file util.c.

Here is the caller graph for this function:



5.30 util.h

Go to the documentation of this file.

```
#ifndef util_h
#define util_h
int RandomInteger(int min, int max);

float RandomFloat(float min, float max);

short RandomFlag(float percentage_chance);

#endif
```

Index