

ecet4640-lab4

Generated by Doxygen 1.9.3

1 ecet4640-lab4	1
1.0.1 About	2
2 Data Structure Index	3
2.1 Data Structures	3
3 File Index	5
3.1 File List	5
4 Data Structure Documentation	7
4.1 map Struct Reference	7
4.1.1 Detailed Description	7
4.1.2 Field Documentation	8
4.1.2.1 buckets	8
4.1.2.2 size	8
4.2 map_result Struct Reference	8
4.2.1 Detailed Description	8
4.2.2 Field Documentation	8
4.2.2.1 data	9
4.2.2.2 found	9
4.3 Student Struct Reference	9
4.3.1 Detailed Description	9
4.3.2 Field Documentation	9
4.3.2.1 active	10
4.3.2.2 age	10
4.3.2.3 fullName	10
4.3.2.4 gpa	10
4.3.2.5 lastLogin	10
4.3.2.6 loginDuration	10
4.3.2.7 userID	10
5 File Documentation	11
5.1 src/server/Build.c File Reference	11
5.1.1 Function Documentation	12
5.1.1.1 BuildStudentMap()	12
5.1.1.2 CalculateCumulative()	13
5.1.1.3 PopulateStudents()	14
5.1.1.4 ProcessWhoLine()	14
5.1.1.5 ReadACP()	15
5.1.1.6 ReadAcpPipeLine()	16
5.1.1.7 ReadCumulativeFileLine()	17
5.1.1.8 ReadInitialCumulative()	18
5.1.1.9 SetAllStudentsInactive()	19
5.1.1.10 UpdateFromWho()	20

5.1.1.11 WriteStudentsToMemory()	20
5.1.2 Variable Documentation	21
5.1.2.1 dirty	21
5.1.2.2 students	21
5.2 Build.c	22
5.3 src/server/Build.h File Reference	24
5.3.1 Function Documentation	26
5.3.1.1 BuildStudentMap()	26
5.3.1.2 CalculateCumulative()	27
5.3.1.3 PopulateStudents()	28
5.3.1.4 ProcessWhoLine()	28
5.3.1.5 ReadACP()	29
5.3.1.6 ReadAcpPipeLine()	30
5.3.1.7 ReadCumulativeFileLine()	31
5.3.1.8 ReadInitialCumulative()	32
5.3.1.9 SetAllStudentsInactive()	33
5.3.1.10 UpdateFromWho()	34
5.3.1.11 WriteStudentsToMemory()	34
5.3.2 Variable Documentation	35
5.3.2.1 dirty	35
5.3.2.2 students	35
5.4 Build.h	36
5.5 src/server/Data.c File Reference	36
5.5.1 Variable Documentation	37
5.5.1.1 Data_IDs	37
5.5.1.2 Data_Names	37
5.6 Data.c	38
5.7 src/server/Data.h File Reference	38
5.7.1 Macro Definition Documentation	39
5.7.1.1 DATA_ID_MAX_LENGTH	39
5.7.1.2 DATA_NAME_MAX_LENGTH	40
5.7.1.3 DATA_NUM_RECORDS	40
5.7.1.4 DATA_SIZE	40
5.7.2 Variable Documentation	40
5.7.2.1 Data_IDs	40
5.7.2.2 Data_Names	40
5.8 Data.h	41
5.9 src/server/Files.c File Reference	41
5.9.1 Function Documentation	42
5.9.1.1 CreateInitialCumulativeFile()	42
5.9.1.2 CreateInitialUserDataFile()	43
5.9.1.3 FileExists()	44

5.9.1.4 FillStudentMapFromFile()	44
5.9.1.5 WriteStudentArrayToFile()	45
5.10 Files.c	46
5.11 src/server/Files.h File Reference	47
5.11.1 Macro Definition Documentation	49
5.11.1.1 STATIC_USER_CUMULATIVE_FILE	49
5.11.1.2 STATIC_USER_DATA_FILE	49
5.11.2 Function Documentation	49
5.11.2.1 CreateInitialCumulativeFile()	49
5.11.2.2 CreateInitialUserDataFile()	50
5.11.2.3 FileExists()	51
5.11.2.4 FillStudentMapFromFile()	52
5.11.2.5 WriteStudentArrayToFile()	53
5.12 Files.h	53
5.13 src/server/main.c File Reference	54
5.13.1 Function Documentation	54
5.13.1.1 main()	55
5.14 main.c	55
5.15 src/server/map.c File Reference	56
5.15.1 Function Documentation	57
5.15.1.1 _bucket_delete()	57
5.15.1.2 _bucket_get()	58
5.15.1.3 _bucket_insert()	58
5.15.1.4 hash_log2()	58
5.15.1.5 hash_string()	59
5.15.1.6 hash_upperLimit()	60
5.15.1.7 Map_Delete()	60
5.15.1.8 Map_Get()	61
5.15.1.9 Map_Set()	62
5.15.1.10 NewMap()	63
5.15.2 Variable Documentation	64
5.15.2.1 char_ratio	64
5.16 map.c	64
5.17 src/server/map.h File Reference	67
5.17.1 Function Documentation	67
5.17.1.1 hash_log2()	68
5.17.1.2 hash_string()	69
5.17.1.3 hash_upperLimit()	70
5.17.1.4 Map_Delete()	71
5.17.1.5 Map_Get()	71
5.17.1.6 Map_Set()	72
5.17.1.7 NewMap()	73

5.18 map.h	74
5.19 src/server/memShare.c File Reference	75
5.19.1 Function Documentation	75
5.19.1.1 CreateSharedMemory()	76
5.19.1.2 DestroySharedMemory()	76
5.19.1.3 GetMemoryPointer()	76
5.19.1.4 ReleaseMemoryPointer()	77
5.20 memShare.c	77
5.21 src/server/memShare.h File Reference	78
5.21.1 Macro Definition Documentation	79
5.21.1.1 MEM_KEY	79
5.21.1.2 MEM_PERMISSIONS	79
5.21.1.3 MEM_SIZE	79
5.21.2 Function Documentation	79
5.21.2.1 CreateSharedMemory()	80
5.21.2.2 DestroySharedMemory()	80
5.21.2.3 GetMemoryPointer()	80
5.21.2.4 ReleaseMemoryPointer()	81
5.22 memShare.h	81
5.23 src/server/Process.c File Reference	82
5.23.1 Function Documentation	83
5.23.1.1 CreateLockfile()	83
5.23.1.2 DeleteLockfile()	84
5.23.1.3 DoesLockfileExist()	84
5.23.1.4 HelpCommand()	85
5.23.1.5 IndicateRereadDone()	86
5.23.1.6 IndicateRereadNeeded()	86
5.23.1.7 Initialize()	87
5.23.1.8 IsRereadNeeded()	88
5.23.1.9 Process()	88
5.23.1.10 ResetCommand()	89
5.23.1.11 RunCommand()	90
5.23.1.12 RunHeadless()	92
5.23.1.13 SignalHandle()	92
5.23.1.14 StopCommand()	93
5.23.1.15 TerminateExistingServer()	94
5.23.2 Variable Documentation	94
5.23.2.1 initial_cumulative_times	94
5.23.2.2 is_stopping	94
5.23.2.3 student_map	95
5.24 Process.c	95
5.25 src/server/Process.h File Reference	99

5.25.1 Macro Definition Documentation	100
5.25.1.1 LOCKFILE	100
5.25.2 Function Documentation	100
5.25.2.1 ClearCommand()	101
5.25.2.2 CreateLockfile()	101
5.25.2.3 DeleteLockfile()	102
5.25.2.4 DoesLockfileExist()	102
5.25.2.5 HelpCommand()	103
5.25.2.6 IndicateRereadDone()	104
5.25.2.7 IndicateRereadNeeded()	104
5.25.2.8 Initialize()	105
5.25.2.9 IsRereadNeeded()	106
5.25.2.10 Process()	106
5.25.2.11 ResetCommand()	107
5.25.2.12 RunCommand()	108
5.25.2.13 RunHeadless()	110
5.25.2.14 SignalHandle()	110
5.25.2.15 StopCommand()	111
5.25.2.16 TerminateExistingServer()	112
5.25.3 Variable Documentation	112
5.25.3.1 is_stopping	112
5.25.3.2 student_map	112
5.26 Process.h	113
5.27 src/server/util.c File Reference	113
5.27.1 Function Documentation	114
5.27.1.1 RandomFlag()	114
5.27.1.2 RandomFloat()	114
5.27.1.3 RandomInteger()	115
5.28 util.c	116
5.29 src/server/util.h File Reference	116
5.29.1 Function Documentation	116
5.29.1.1 RandomFlag()	117
5.29.1.2 RandomFloat()	117
5.29.1.3 RandomInteger()	118
5.30 util.h	119

Index	121
--------------	------------

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:

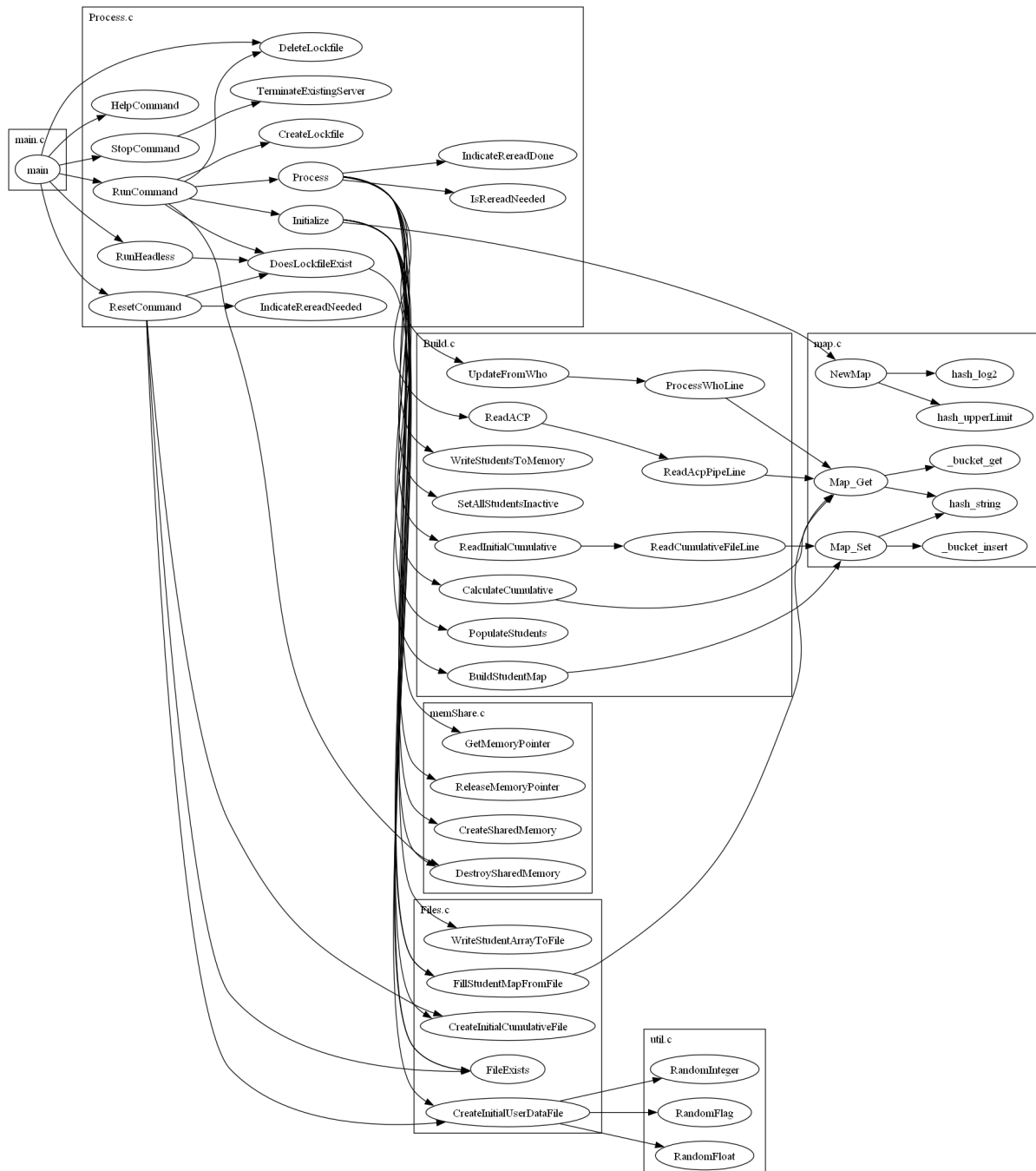


Figure 1.1 Program Call graph

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:

map	A map. Stores key-value pairs for near constant lookup and insertion time	7
map_result	The result of a map retrieval	8
Student	The student data type	9

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

src/server/Build.c	11
src/server/Build.h	24
src/server/Data.c	36
src/server/Data.h	38
src/server/Files.c	41
src/server/Files.h	47
src/server/main.c	54
src/server/map.c	56
src/server/map.h	67
src/server/memShare.c	75
src/server/memShare.h	78
src/server/Process.c	82
src/server/Process.h	99
src/server/util.c	113
src/server/util.h	116

Chapter 4

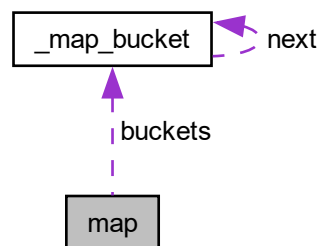
Data Structure Documentation

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](#)

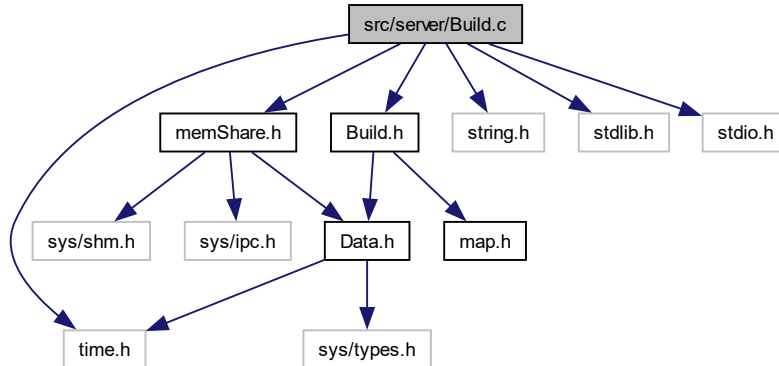
Chapter 5

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 *stud_arr, int stud_arrLength)
- 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 )
```

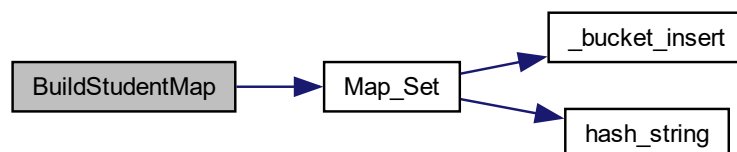
Given a student array, populates a student map, where the student IDs are the key, and the values are pointers to the items in the array.

Parameters

<i>map</i>	The map structure to populate.
<i>studentArr</i>	An array of student structures.
<i>studentArrLength</i>	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

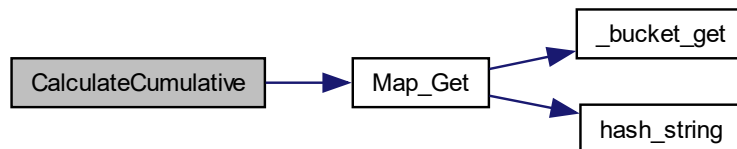
student.loginDuration must have already been set to the total cumulative time logged in.

Parameters

<i>stud_arr</i>	The student's array.
<i>arr_len</i>	The length of students array.
<i>cum_map</i>	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

<i>studentIDs</i>	An array of student IDs.
<i>studentNames</i>	An array of student names.
<i>arsize</i>	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

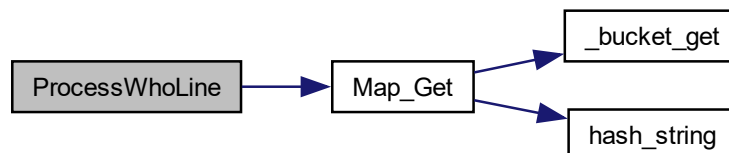
<i>stmap</i>	The student map.
<i>whoLine</i>	The line of text, such as returned from <code>fgets</code>
<i>whoLineLength</i>	The length of that text.

Returns

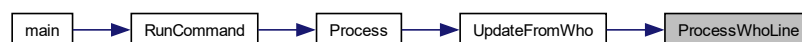
0 if success, -1 if the student was not found in the map.

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

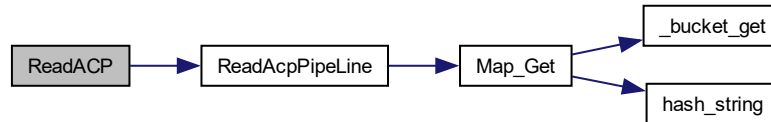
<code>st_map</code>	The students map.
---------------------	-------------------

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

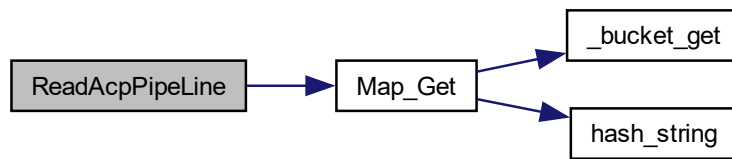
<i>stmap</i>	A map of students.
<i>acp_line</i>	A string representing 1 line result from <code>ac -p</code> .

Returns

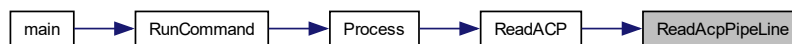
-1 if `acp_line` is NULL or length is less than 1, otherwise 0.

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

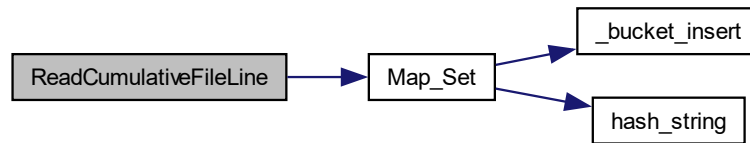
<i>cum_map</i>	The cumulative map.
<i>acp_line</i>	A single line from <code>ac -p</code> .

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

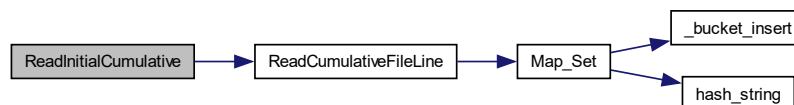
<i>cum_map</i>	A map of cumulative times. Different from the students map.
<i>filename</i>	The filename where the initial cumulative times are located.

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

<i>stud_arr</i>	The students array.
<i>arr_len</i>	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

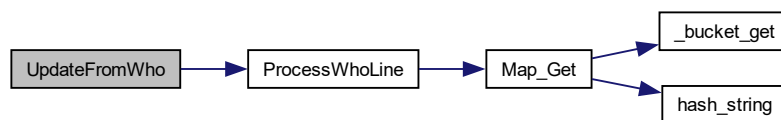
<i>stmap</i>	The student map.
--------------	------------------

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 )
```

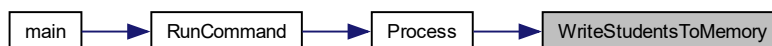
Writes the students array to the location specified by `mem_ptr` (eg. the shared memory segment).

Parameters

<i>mem_ptr</i>	The address to write at.
<i>stud_arr</i>	The students array to write.
<i>arr_len</i>	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.](#)

```

00001
00004 #include "Build.h"
00005 #include "memShare.h"
00006 #include <string.h>
00007 #include <stdlib.h>
00008 #include <stdio.h>
00009 #include <time.h>
00010
00011 // ~~~~~ Data Structures ~~~~~
00012
00013 Student *students;
00014
00015 void PopulateStudents(char **studentIDs, char **studentNames, int arsize)
00016 {
00017     students = malloc(sizeof(Student) * arsize);
00018     int i;
00019     for (i = 0; i < arsize; i++)
00020     {
00021         strcpy(students[i].userID, studentIDs[i]);
00022         strcpy(students[i].fullName, studentNames[i]);
00023         // students[i].age = randAge(18, 22);
00024     }
00025 }
00026
00027 void BuildStudentMap(map *stmap, Student *studentArr, int studentArrLength)
00028 {
00029     int i;
00030     for (i = 0; i < studentArrLength; i++)
00031     {
00032         Map_Set(stmap, studentArr[i].userID, (void *)&studentArr[i]);
00033     }
00034 }
00035
00036 // ~~~~~ Processing ~~~~~
00037
00038 short dirty = 1; // start dirty
00039
00040 int UpdateFromWho(map *stmap)
00041 {
00042     char command[4] = "who";
00043     char line[100];
00044     FILE *fpipe;
00045     fpipe = popen(command, "r");
00046     if (fpipe == NULL)
00047     {
00048         return -1;
00049     }
00050
00051     while (fgets(line, sizeof(line), fpipe) != NULL)
00052     {
00053         ProcessWhoLine(stmap, line, strlen(line));
00054     }
00055     pclose(fpipe);
00056
00057     return 0;
00058 }
00059
00060 int ProcessWhoLine(map *stmap, char *whoLine, int whoLineLength)
00061 {
00062     char userId[20];
00063     char dateString[50];
00064     char timeString[20];
00065     int read_total = 0;
00066     int read;
00067     sscanf(whoLine, "%s %n", userId, &read);
00068     read_total += read;
00069
00070     map_result mr = Map_Get(stmap, userId);
00071     if (!mr.found)
00072     { // if we can't find that person in the map, return early
00073         return -1;
00074     }
00075     Student *student = (Student *)mr.data;
00076
00077     sscanf(whoLine + read_total, "%s %n", dateString, &read); // will be thrown away. eg 'pts/1'
00078     read_total += read;
00079     sscanf(whoLine + read_total, "%s %n", dateString, &read); // read the date string
00080     read_total += read;
00081     sscanf(whoLine + read_total, "%s %n", timeString, &read); // read the time string
00082     strcat(dateString, " ");
00083     strcat(dateString, timeString); // concatenate the time string back to the date string
00084

```

```

00085     time_t now = time(NULL);
00086     struct tm dtime = *localtime(&now);
00087     dtime.tm_sec = 0;
00088
00089     memset(&dtime, 0, sizeof(struct tm));
00090
00091     sscanf(dateString, "%d-%d-%d %d:%d", &(dtime.tm_year), &(dtime.tm_mon), &(dtime.tm_mday),
00092            &(dtime.tm_hour), &(dtime.tm_min));
00093     dtime.tm_year -= 1900;
00094     dtime.tm_mon -= 1;
00095     dtime.tm_hour -= 1;
00096
00097     time_t parsed_time = mktime(&dtime);
00098
00099     if (student->lastLogin != parsed_time)
00100     {
00101         student->lastLogin = parsed_time;
00102         dirty = 1;
00103     }
00104     student->active = 1;
00105     return 0;
00106 }
00107
00108 void SetAllStudentsInactive(Student *stud_arr, int arr_len)
00109 {
00110     int i;
00111     for (i = 0; i < arr_len; i++)
00112     {
00113         stud_arr[i].active = 0;
00114     }
00115 }
00116
00117 void WriteStudentsToMemory(void *mem_ptr, Student *stud_arr, int arr_len)
00118 {
00119     Student *memloc = (Student *)mem_ptr;
00120     int i;
00121     for (i = 0; i < arr_len; i++)
00122     {
00123         strcpy(memloc[i].userID, stud_arr[i].userID);
00124         strcpy(memloc[i].fullName, stud_arr[i].fullName);
00125         memloc[i].age = stud_arr[i].age;
00126         memloc[i].gpa = stud_arr[i].gpa;
00127         memloc[i].active = stud_arr[i].active;
00128         memloc[i].lastLogin = stud_arr[i].lastLogin;
00129         memloc[i].loginDuration = stud_arr[i].loginDuration;
00130     }
00131 }
00132
00133 // ~~~~~ Cumulative Processing ~~~~~
00134
00135 int ReadInitialCumulative(map *time_map, char *filename)
00136 {
00137     FILE *file = fopen(filename, "r");
00138     char line[100];
00139     if (file == NULL)
00140     {
00141         return -1;
00142     }
00143     while (fgets(line, sizeof(line), file) != NULL)
00144     {
00145         ReadCumulativeFileLine(time_map, line);
00146     }
00147
00148     fclose(file);
00149     return 0;
00150 }
00151
00152 int ReadACP(map *st_map)
00153 {
00154     char command[6] = "ac -p";
00155     char line[300];
00156     FILE *fpipe;
00157     fpipe = popen(command, "r");
00158     if (fpipe == NULL)
00159     {
00160         return -1;
00161     }
00162     int err;
00163     while (fgets(line, sizeof(line), fpipe) != NULL)
00164     {
00165         err = ReadAcpPipeline(st_map, line);
00166         if (err)
00167         {
00168             printf("\nError %d reading acp pipeline.", err);
00169             break;
00170         }
00171     }

```

```

00171     }
00172     pclose(fpipeline);
00173     return 0;
00174 }
00175
00176 void ReadCumulativeFileLine(map *cum_map, char *acp_line)
00177 {
00178     char userId[20];
00179     float minutes;
00180     sscanf(acp_line, " %s %f ", userId, &minutes);
00181     // int seconds = (int) (minutes * 60)
00182     long seconds = (long) (minutes * 60);
00183     Map_Set(cum_map, userId, (void *)seconds);
00184 }
00185
00186 int ReadAcpPipeLine(map *stmap, char *acp_line)
00187 {
00188     if (acp_line == NULL || strlen(acp_line) < 1)
00189     {
00190         return -1;
00191     }
00192     char userId[40];
00193     float minutes;
00194     sscanf(acp_line, "%s %f", userId, &minutes);
00195     map_result result = Map_Get(stmap, userId);
00196     if (result.found)
00197     {
00198         Student *student = (Student *)result.data;
00199         int seconds = (int) (minutes * 60);
00200         student->loginDuration = seconds;
00201     }
00202     return 0;
00203 }
00204
00205 void CalculateCumulative(Student *stud_arr, int stud_arr_len, map *cum_map)
00206 {
00207     int i;
00208     for (i = 0; i < stud_arr_len; i++)
00209     {
00210         map_result result = Map_Get(cum_map, stud_arr[i].userID);
00211         if (result.found)
00212         {
00213             long time_at_server_start = (long)result.data;
00214             stud_arr[i].loginDuration = stud_arr[i].loginDuration - time_at_server_start;
00215         }
00216     }
00217 }

```

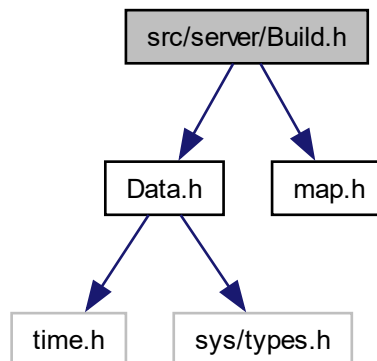
5.3 src/server/Build.h File Reference

```

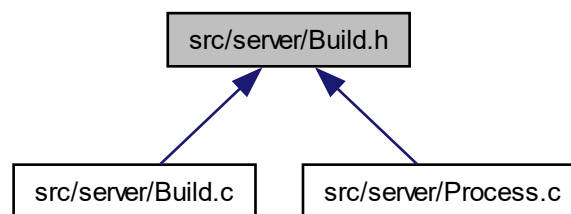
#include "Data.h"
#include "map.h"

```


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 )
```

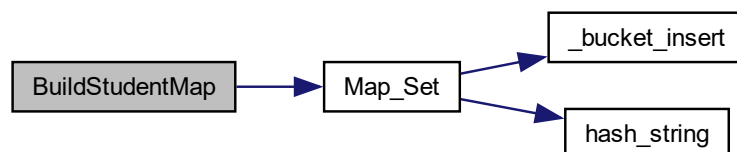
Given a student array, populates a student map, where the student IDs are the key, and the values are pointers to the items in the array.

Parameters

<i>map</i>	The map structure to populate.
<i>studentArr</i>	An array of student structures.
<i>studentArrLength</i>	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

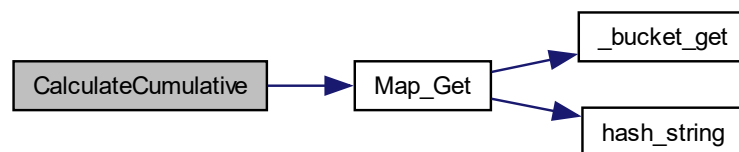
student.loginDuration must have already been set to the total cumulative time logged in.

Parameters

<i>stud_arr</i>	The student's array.
<i>arr_len</i>	The length of students array.
<i>cum_map</i>	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

<i>studentIDs</i>	An array of student IDs.
<i>studentNames</i>	An array of student names.
<i>arsize</i>	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

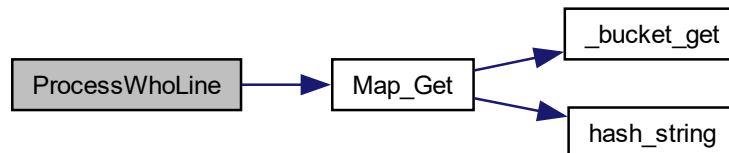
<i>stmap</i>	The student map.
<i>whoLine</i>	The line of text, such as returned from <code>fgets</code>
<i>whoLineLength</i>	The length of that text.

Returns

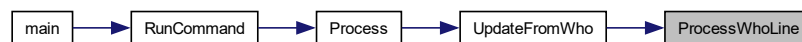
0 if success, -1 if the student was not found in the map.

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

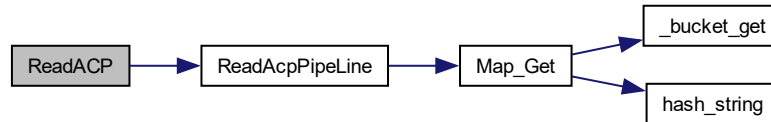
<code>st_map</code>	The students map.
---------------------	-------------------

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

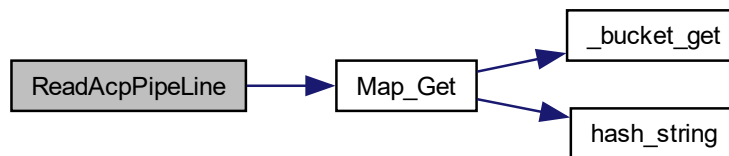
<i>stmap</i>	A map of students.
<i>acp_line</i>	A string representing 1 line result from <code>ac -p</code> .

Returns

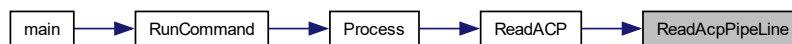
-1 if `acp_line` is NULL or length is less than 1, otherwise 0.

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

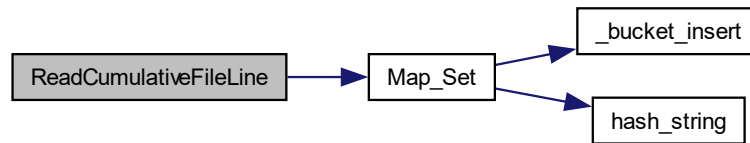
<i>cum_map</i>	The cumulative map.
<i>acp_line</i>	A single line from <code>ac -p</code> .

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

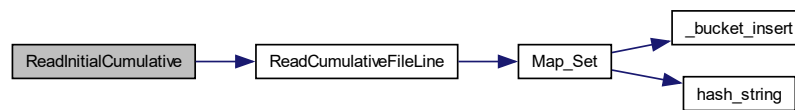
<i>cum_map</i>	A map of cumulative times. Different from the students map.
<i>filename</i>	The filename where the initial cumulative times are located.

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

<i>stud_arr</i>	The students array.
<i>arr_len</i>	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

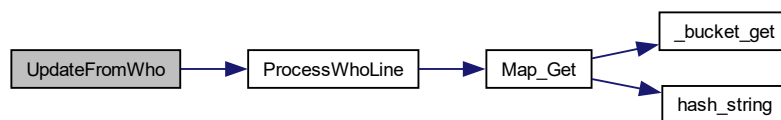
<i>stmap</i>	The student map.
--------------	------------------

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 )
```

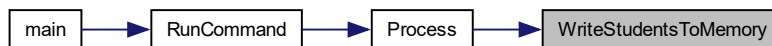
Writes the students array to the location specified by `mem_ptr` (eg. the shared memory segment).

Parameters

<i>mem_ptr</i>	The address to write at.
<i>stud_arr</i>	The students array to write.
<i>arr_len</i>	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.](#)

```

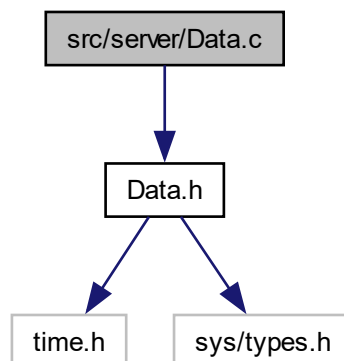
00001 #ifndef BUILD_H
00002 #define BUILD_H
00007 #include "Data.h"
00008 #include "map.h"
00009
00010 // ~~~~~ Data Structures ~~~~~
00011
00017 extern Student *students;
00018
00026 void PopulateStudents(char **studentIDs, char **studentNames, int arsize);
00027
00034 void BuildStudentMap(map *stmap, Student *studentArr, int studentArrLength);
00035
00036 // ~~~~~ Processing ~~~~~
00037
00039 extern short dirty;
00040
00047 int UpdateFromWho(map *stmap);
00048
00062 int ProcessWhoLine(map *stmap, char *whoLine, int whoLineLength);
00063
00070 void SetAllStudentsInactive(Student *stud_arr, int arr_len);
00071
00080 void WriteStudentsToMemory(void *mem_ptr, Student *stud_arr, int arr_len);
00081
00082 // ~~~~~ Cumulative Processing ~~~~~
00083
00093 int ReadInitialCumulative(map *time_map, char *filename);
00094
00103 int ReadACP(map *st_map);
00104
00114 void ReadCumulativeFileLine(map *cum_map, char *acp_line);
00115
00123 int ReadAcpPipeLine(map *stmap, char *acp_line);
00124
00134 void CalculateCumulative(Student *stud_arr, int stud_arr_len, map *cum_map);
00135
00136 #endif

```

5.5 src/server/Data.c File Reference

```
#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",  
    "beal389",  
    "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](#).

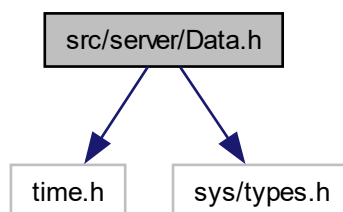
5.6 Data.c

[Go to the documentation of this file.](#)

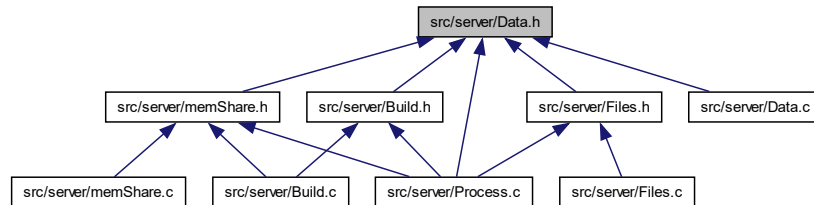
```
00001
00004 #include "Data.h"
00005
00006 char *Data_IDs[DATA_NUM_RECORDS] = {
00007     "chen",
00008     "beal389",
00009     "bol4559",
00010     "cal6258",
00011     "kre5277",
00012     "lon1150",
00013     "mas9309",
00014     "mes08346",
00015     "mil7233",
00016     "nef9476",
00017     "Nov-88",
00018     "pan9725",
00019     "rac3146",
00020     "rub4133",
00021     "shr5683",
00022     "vay3083",
00023     "yos2327"};
00024
00025 char *Data_Names[DATA_NUM_RECORDS] = {
00026     "Weifeng Chen",
00027     "Christian Beatty",
00028     "Emily Bolles",
00029     "Cameron Calhoun",
00030     "Ty Kress",
00031     "Cody Long",
00032     "Caleb Massey",
00033     "Christian Messmer",
00034     "Karl Miller",
00035     "Jeremiah Neff",
00036     "Kaitlyn Novacek",
00037     "Joshua Panaro",
00038     "Caleb Rachocki",
00039     "Caleb Ruby",
00040     "Paul Shriner",
00041     "Alan Vayansky",
00042     "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.](#)

```

00001 #ifndef Data_h
00002 #define Data_h
00006 #include <time.h>
00007 #include <sys/types.h>
00008
00012 #define DATA_NUM_RECORDS 17
00017 #define DATA_ID_MAX_LENGTH 9
00022 #define DATA_NAME_MAX_LENGTH 21
00023
00024 /* Constant, all user IDs. */
00025 extern char *Data_IDs[];
00026
00028 extern char *Data_Names[];
00029
00033 #define DATA_SIZE 56
00034
00038 typedef struct
00039 {
00040     char userID[DATA_ID_MAX_LENGTH];
00041     char fullName[DATA_NAME_MAX_LENGTH];
00042     short age;
00043     float gpa;
00044     short active;
00045     time_t lastLogin;
00046     int loginDuration;
00047 } Student;
00048
00049 #endif

```

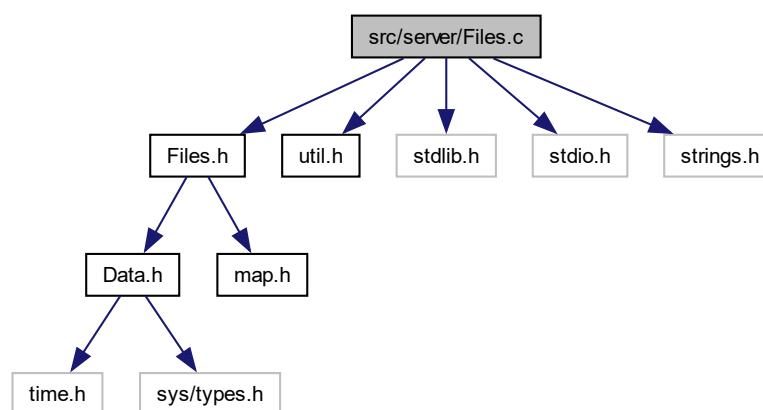
5.9 src/server/Files.c File Reference

```

#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

<i>file_name</i>	The name of the file to created. EG STATIC_USER_CUMULATIVE_FILE
------------------	---

Warning

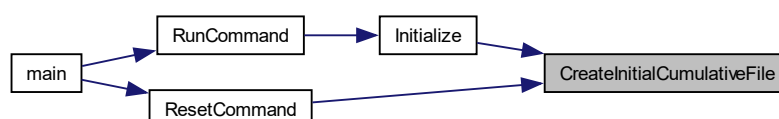
This file should already be validated to not exist.

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

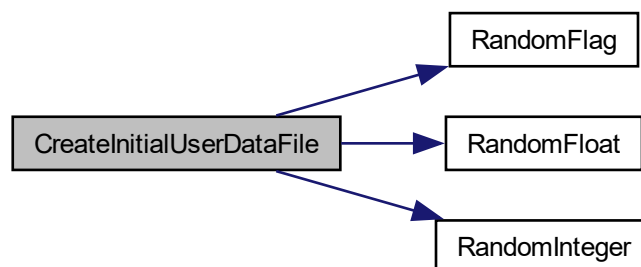
<i>file_name</i>	The file name to create.
<i>id_list</i>	An array containing the IDs. Eg. "Data_IDs" from Data.h
<i>id_list_len</i>	The length of the id_list. Eg. "DATA_NUM_RECORDS" from Data.h

Returns

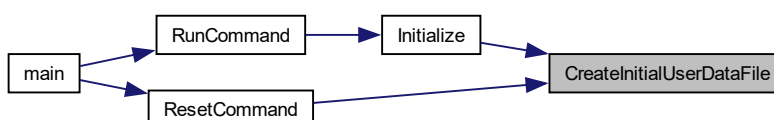
A 0 if the operation was succesful, otherwise nonzero.

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()

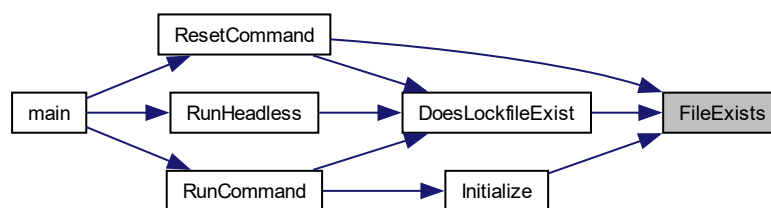
```
short FileExists (
    char * file_name_to_check )
```

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

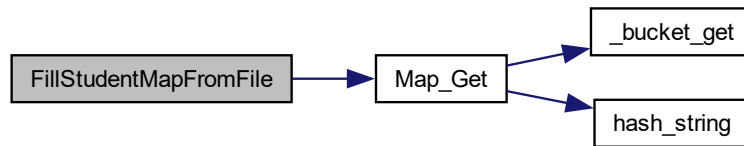
<i>student_map</i>	The map of student structs to be populated from the login.txt file
<i>file_name</i>	The name of the login.txt file.
<i>id_list</i>	An array containing the IDs. Eg. "Data_IDs" from Data.h
<i>id_list_len</i>	The length of the id_list. Eg. "DATA_NUM_RECORDS" from Data.h

Returns

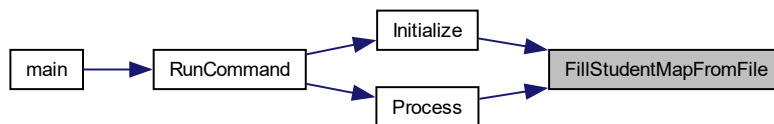
0 if succesful, 1 if there was an error.

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

<i>students</i>	A pointer to the student array that will be read into the file.
<i>arr_len</i>	The length of the students array. e.g. DATA_NUM_RECORDS from Data.h .
<i>file_name</i>	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.](#)

```

00001
00004 #include "Files.h"
00005 #include "util.h"
00006 #include <stdlib.h>
00007 #include <stdio.h>
00008 #include <strings.h>
00009
00010 short FileExists(char *file_name_to_check)
00011 {
00012     FILE *file = fopen(file_name_to_check, "r");
00013     short result = 1;
00014     if (file == NULL)
00015     {
00016         result = 0;
00017     }
00018     else
00019     {
00020         fclose(file);
00021     }
00022     return result;
00023 }
00024
00025 int CreateInitialUserDataFile(char *file_name, char **id_list, int id_list_len)
00026 {
00027     FILE *file = fopen(file_name, "w");
00028     if (file == NULL)
00029     {
00030         return -1;
00031     }
00032     int i;
00033     for (i = 0; i < id_list_len; i++)
00034     {
00035         int rand_age = RandomInteger(18, 22);
00036         float gpa;
00037         if (RandomFlag(0.42))
00038         {
00039             gpa = 4.0; // 42% of the time, make the GPA 4.0
00040         }
00041         else
00042         {
00043             gpa = RandomFloat(2.5, 4.0);
00044         }
00045         fprintf(file, "%s\t%d\t%.2f\t%d\n", id_list[i], rand_age, gpa, 0);
00046     }
00047     fclose(file);
00048     return 0;
00049 }
00050
00051 int FillStudentMapFromFile(map *student_map, char *file_name, char **id_list, int id_list_len)
00052 {
00053     FILE *file = fopen(file_name, "r");
00054     if (file == NULL)
00055     {
00056         return -1;
00057     }
00058     // id buffer
00059     char user_id[9];
00060     int age;
00061     float gpa;
00062     long time;
00063     while (fscanf(file, "%9s\t%d\t%f\t%d", user_id, &age, &gpa, &time) == 4)
00064     {
00065         map_result result = Map_Get(student_map, user_id);
00066         if (result.found == 0)

```

```

00067     {
00068         continue;
00069     }
00070     ((Student *)result.data)->age = age;
00071     ((Student *)result.data)->gpa = gpa;
00072     ((Student *)result.data)->lastLogin = time;
00073 }
00074 fclose(file);
00075 return 0;
00076 }
00077
00078 int WriteStudentArrayToFile(Student *students, int arr_len, char *file_name)
00079 {
00080     FILE *file = fopen(file_name, "w");
00081     if (file == NULL)
00082     {
00083         return -1;
00084     }
00085     int i;
00086     for (i = 0; i < arr_len; i++)
00087     {
00088         fprintf(file, "%s\t%d\t%.2f\t%d\n", students[i].userID, students[i].age, students[i].gpa,
00089             students[i].lastLogin);
00089     }
00090     fclose(file);
00091     return 0;
00092 }
00093
00094 int CreateInitialCumulativeFile(char *file_name)
00095 {
00096     FILE *file = fopen(file_name, "w");
00097     if (file == NULL)
00098     {
00099         return -1;
00100     }
00101     FILE *pipe = popen("ac -p", "r");
00102     if (pipe == NULL)
00103     {
00104         fclose(file);
00105         return -2;
00106     }
00107
00108     char line[100];
00109     while (fgets(line, sizeof(line), pipe) != NULL)
00110     {
00111         fputs(line, file);
00112     }
00113     pclose(pipe);
00114     fclose(file);
00115     return 0;
00116 }

```

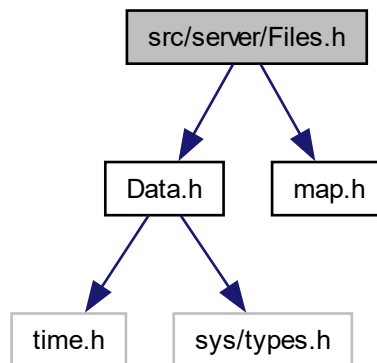
5.11 src/server/Files.h File Reference

```

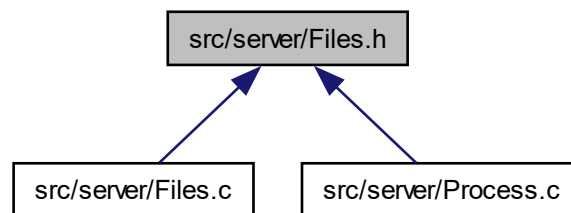
#include "Data.h"
#include "map.h"

```

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

<i>file_name</i>	The name of the file to created. EG STATIC_USER_CUMULATIVE_FILE
------------------	---

Warning

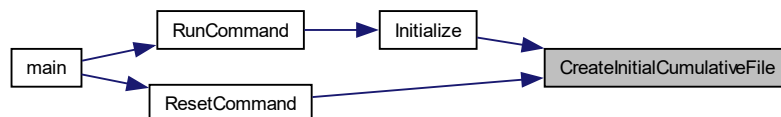
This file should already be validated to not exist.

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

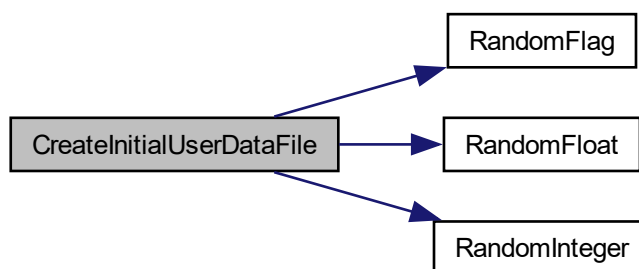
<i>file_name</i>	The file name to create.
<i>id_list</i>	An array containing the IDs. Eg. "Data_IDs" from Data.h
<i>id_list_len</i>	The length of the id_list. Eg. "DATA_NUM_RECORDS" from Data.h

Returns

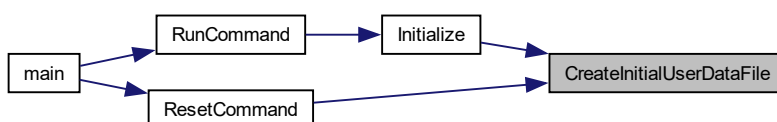
A 0 if the operation was succesful, otherwise nonzero.

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.

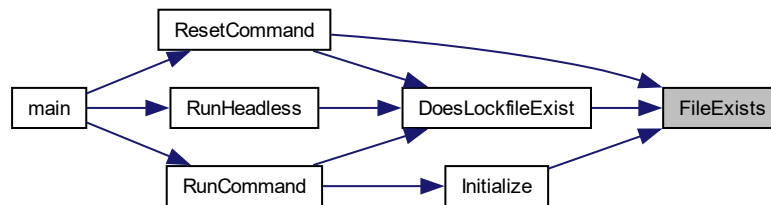
Returns

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

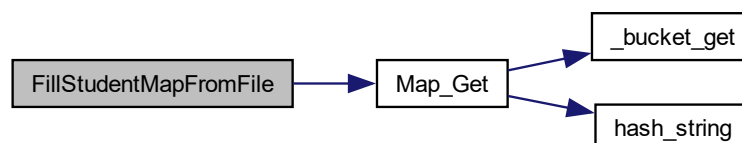
<i>student_map</i>	The map of student structs to be populated from the login.txt file
<i>file_name</i>	The name of the login.txt file.
<i>id_list</i>	An array containing the IDs. Eg. "Data_IDs" from Data.h
<i>id_list_len</i>	The length of the id_list. Eg. "DATA_NUM_RECORDS" from Data.h

Returns

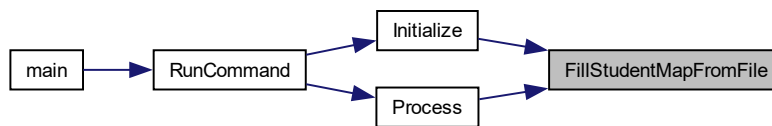
0 if succesful, 1 if there was an error.

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

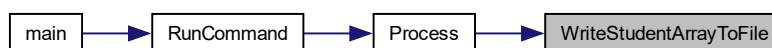
<i>students</i>	A pointer to the student array that will be read into the file.
<i>arr_len</i>	The length of the students array. e.g. DATA_NUM_RECORDS from Data.h .
<i>file_name</i>	The file name to write.

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

```

```

00006 #include "Data.h"
00007 #include "map.h"
00008
00031 #define STATIC_USER_DATA_FILE "static-user-data.txt"
00032
00038 #define STATIC_USER_CUMULATIVE_FILE "static-user-cumulative-start.txt"
00039
00044 short FileExists(char *file_name_to_check);
00045
00055 int CreateInitialUserDataFile(char *file_name, char **id_list, int id_list_len);
00056
00066 int WriteStudentArrayToFile(Student *students, int arr_len, char *file_name);
00067
00078 int FillStudentMapFromFile(map *student_map, char *file_name, char **id_list, int id_list_len);
00079
00089 int CreateInitialCumulativeFile(char *file_name);
00090
00091 #endif

```

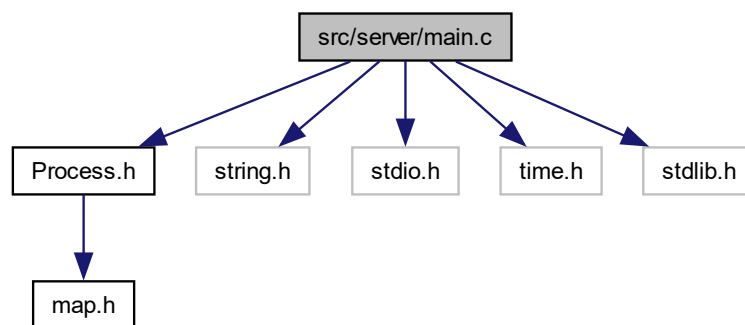
5.13 src/server/main.c File Reference

```

#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.

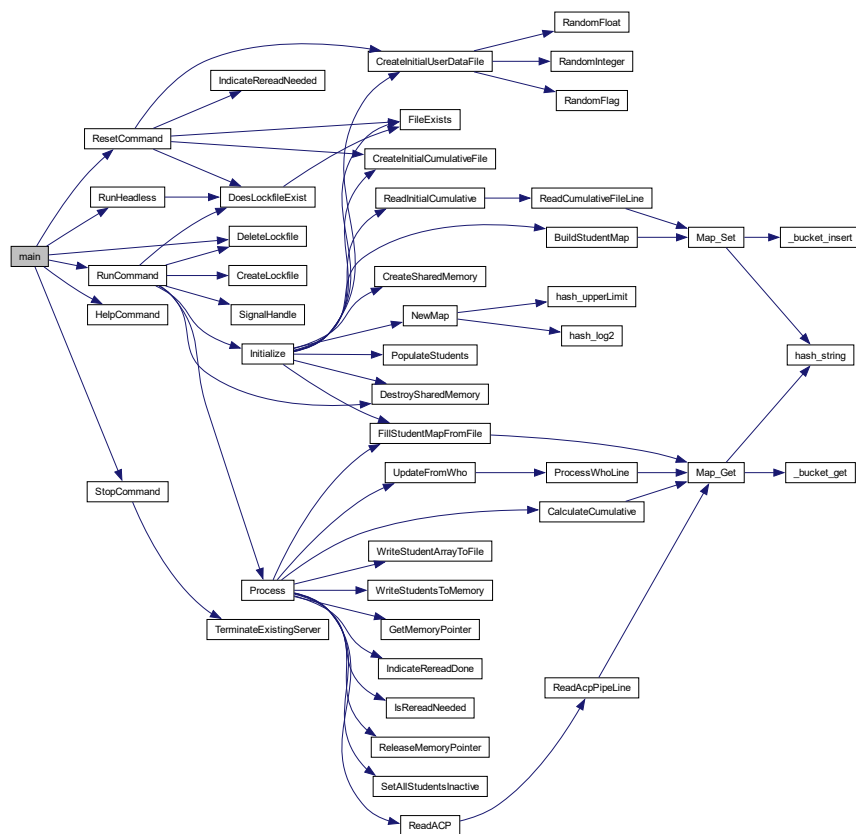
Parses arguments and calls the appropriate [Process.h](#) function.

Parameters

<i>argc</i>	The argument count.
<i>argv</i>	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>
```

```

00003 #include <stdio.h>
00004 #include <time.h>
00005 #include <stdlib.h>
00030 int main(int argc, char **argv)
00031 {
00032     srand(time(NULL)); // seed the randomizer
00033
00034     if (argc <= 1 || argc >= 3)
00035     {
00036         printf("You entered too few or many options!\n");
00037         HelpCommand();
00038     }
00039     else if (strcmp(argv[1], "help") == 0)
00040     {
00041         HelpCommand();
00042     }
00043     else if (strcmp(argv[1], "reset") == 0)
00044     {
00045         ResetCommand();
00046     }
00047     else if (strcmp(argv[1], "stop") == 0 || strcmp(argv[1], "end") == 0 || strcmp(argv[1], "close")
== 0)
00048     {
00049         StopCommand();
00050     }
00051     else if (strcmp(argv[1], "headless") == 0)
00052     {
00053         RunHeadless(argv[0]);
00054     }
00055     else if (strcmp(argv[1], "run") == 0 || strcmp(argv[1], "start") == 0)
00056     {
00057         RunCommand();
00058     }
00059     else if (strcmp(argv[1], "delete-lockfile") == 0)
00060     { // an admin debug command; not meant to be called
00061         printf("Deleting lockfile.\n");
00062         DeleteLockfile();
00063     }
00064     else
00065     {
00066         printf("Unknown option!\n");
00067         HelpCommand();
00068     }
00069     return 0;
00070 }

```

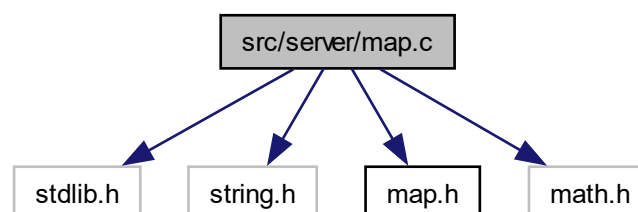
5.15 src/server/map.c File Reference

```

#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 unsuccessful.
- 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:

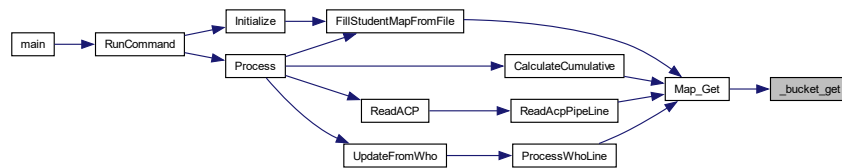


5.15.1.2 `_bucket_get()`

```
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:

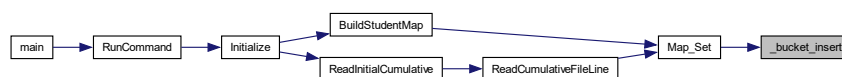


5.15.1.3 `_bucket_insert()`

```
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()`

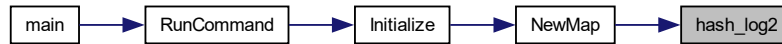
```
int hash_log2 (
    int num_to_log )
```

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 modulusd 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

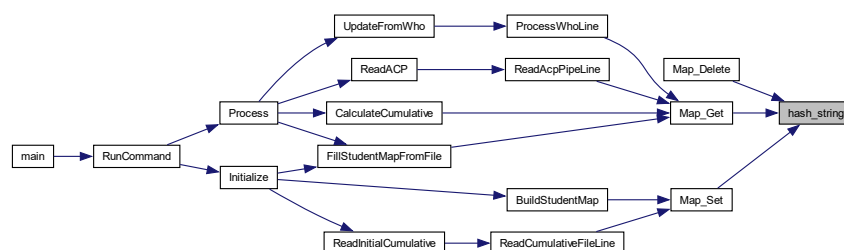
<i>hash_table_capacity</i>	The number of buckets this table holds.
<i>string</i>	The key to hash.
<i>strlen</i>	The length of the key.

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

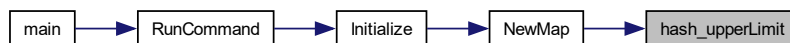
<i>bitsize</i>	The number of bits to calculate the max from.
----------------	---

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

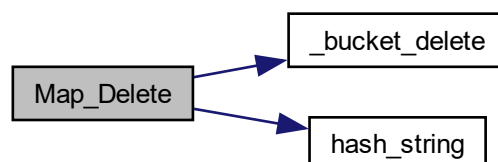
<i>map</i>	The map to delete the key from.
<i>key</i>	The key to delete.
<i>free_it</i>	Whether to call <code>free()</code> on the underlying data.

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 unsuccessful.

Parameters

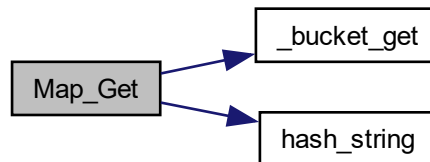
<i>map</i>	The map to retrieve from.
<i>key</i>	The key of the item.

Returns

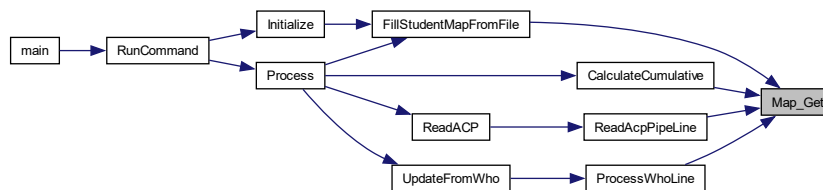
A `map_get_result` containing the sought data.

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 )
  
```

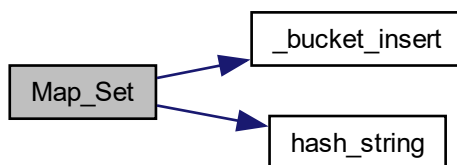
Sets a value in the map.

Parameters

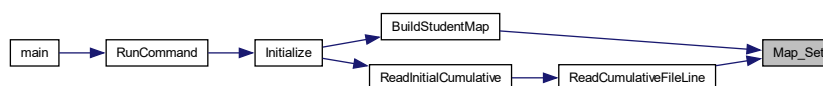
<i>map</i>	The map to set a key in.
<i>key</i>	The key to use.
<i>keylen</i>	The length of the key.
<i>value</i>	The pointer to the data stored at that location.

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

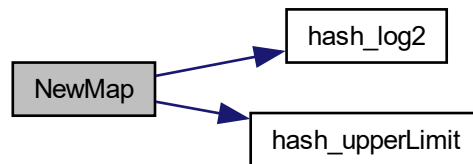
<i>capacity</i>	The estimated required capacity of the map.
-----------------	---

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;
  
```



```

00020 }
00021
00022 int hash_upperLimit(int bitsize)
00023 {
00024     return 1 « bitsize;
00025 }
00026
00027 int char_ratio = (int)(sizeof(int) / sizeof(char));
00028
00029 // Modified some stuff from : http://isthe.com/chongo/tech/comp/fnv/
00030 int hash_string(int hash_table_size, char *string, int strlen)
00031 {
00032     int i, hash = 2166136261;
00033     for (i = 0; i < strlen; i += 1)
00034     {
00035         hash *= 16777619;
00036         hash ^= string[i];
00037     }
00038     if (hash < 0)
00039     {
00040         hash *= -1;
00041     }
00042     return hash % (hash_table_size - 1) + 1;
00043 }
00044
00045 map *NewMap(int capacity)
00046 {
00047     int log2 = hash_log2(capacity);
00048     int capac = hash_upperLimit(log2);
00049     int sz = sizeof(struct _map_bucket) * capac;
00050     struct _map_bucket *buckets = malloc(sz);
00051     memset(buckets, 0, sz);
00052     int i;
00053     for (i = 0; i < capac; i++)
00054     {
00055         buckets[i] = (struct _map_bucket){NULL, NULL, NULL};
00056     }
00057     map newm = (map){capac, buckets};
00058     map *map_p = malloc(sizeof(map));
00059     *map_p = newm;
00060     return map_p;
00061 }
00062
00063 void _bucket_insert(struct _map_bucket *bucket, char *key, void *value)
00064 {
00065     struct _map_bucket *check = bucket;
00066     while (check->key != NULL)
00067     {
00068         if (strcmp(check->key, key) == 0)
00069         {
00070             check->data = value;
00071             return;
00072         }
00073         if (check->next == NULL)
00074         {
00075             check->next = malloc(sizeof(struct _map_bucket));
00076             *(check->next) = (struct _map_bucket){NULL, NULL, NULL};
00077         }
00078         check = check->next;
00079     }
00080     check->key = key;
00081     check->data = value;
00082 }
00083
00084 void Map_Set(map *a_map, char *key, void *value)
00085 {
00086     int keyl = (int)strlen(key);
00087     int hash = hash_string(a_map->size, key, keyl);
00088     _bucket_insert(&(a_map->buckets[hash]), key, value);
00089 }
00090
00091 void _bucket_get(struct _map_bucket *bucket, char *key, map_result *result)
00092 {
00093     struct _map_bucket *check = bucket;
00094     while (check->key != NULL)
00095     {
00096         if (strcmp(check->key, key) == 0)
00097         {
00098             result->found = 1;
00099             result->data = check->data;
00100             return;
00101         }
00102         else if (check->next != NULL)
00103         {
00104             check = check->next;
00105         }
00106         else

```

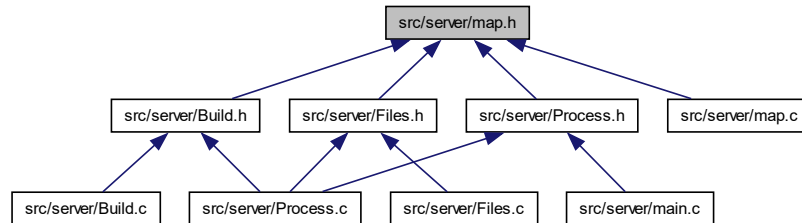
```

00107     {
00108         result->found = 0;
00109         break;
00110     }
00111 }
00112 }
00113
00114 map_result Map_Get(map *a_map, char *key)
00115 {
00116     map_result res = (map_result){0, NULL};
00117     int keyl = (int)strlen(key);
00118     int hash = hash_string(a_map->size, key, keyl);
00119     _bucket_get(&(a_map->buckets[hash]), key, &res);
00120     return res;
00121 }
00122
00123 void _bucket_delete(struct _map_bucket *bucket, char *key, short free_it, map_result *result)
00124 {
00125     struct _map_bucket *last = bucket;
00126     struct _map_bucket *next = bucket->next;
00127     while (next != NULL)
00128     {
00129         if (strcmp(next->key, key) == 0)
00130         {
00131             result->found = 1;
00132             result->data = next->data;
00133             if (free_it)
00134             {
00135                 free(next->data);
00136                 result->data = NULL;
00137             }
00138             last->next = next->next;
00139             free(next);
00140         }
00141         else
00142         {
00143             last = next;
00144             next = next->next;
00145         }
00146     }
00147 }
00148
00149 map_result Map_Delete(map *a_map, char *key, short free_it)
00150 {
00151     map_result res = (map_result){0, NULL};
00152     int keyl = (int)strlen(key);
00153     int hash = hash_string(a_map->size, key, keyl);
00154
00155     struct _map_bucket top = a_map->buckets[hash];
00156     if (top.key == NULL)
00157     {
00158         return res;
00159     }
00160     if (strcmp(top.key, key) == 0)
00161     {
00162         res.found = 1;
00163         res.data = top.data;
00164         if (free_it)
00165         {
00166             free(top.data);
00167             res.data = NULL;
00168         }
00169         if (top.next != NULL)
00170         {
00171             a_map->buckets[hash] = *(top.next);
00172             free(top.next);
00173         }
00174         else
00175         {
00176             a_map->buckets[hash] = (struct _map_bucket){NULL, NULL, NULL};
00177         }
00178         return res;
00179     }
00180     if (top.next == NULL)
00181     {
00182         return res;
00183     }
00184     _bucket_delete(&(a_map->buckets[hash]), key, free_it, &res);
00185
00186     return res;
00187 }

```

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 unsuccessful.
- [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

<i>number_to_log</i>	The number to calculate the log of.
----------------------	-------------------------------------

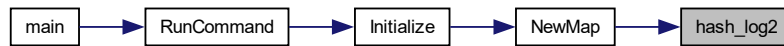
Returns

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

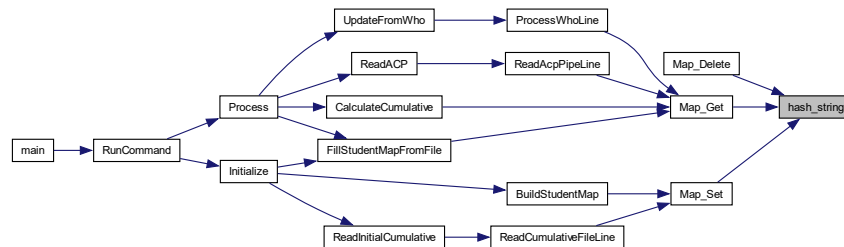
<i>hash_table_capacity</i>	The number of buckets this table holds.
<i>string</i>	The key to hash.
<i>strlen</i>	The length of the key.

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

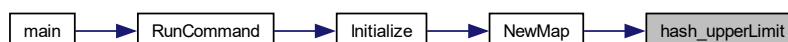
<i>bitsize</i>	The number of bits to calculate the max from.
----------------	---

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

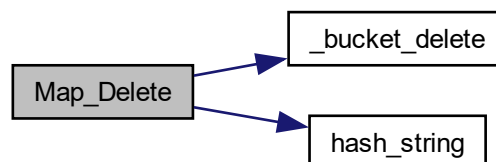
<i>map</i>	The map to delete the key from.
<i>key</i>	The key to delete.
<i>free_it</i>	Whether to call <code>free()</code> on the underlying data.

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 unsuccessful.

Parameters

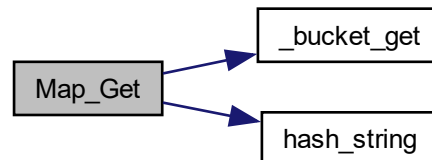
<i>map</i>	The map to retrieve from.
<i>key</i>	The key of the item.

Returns

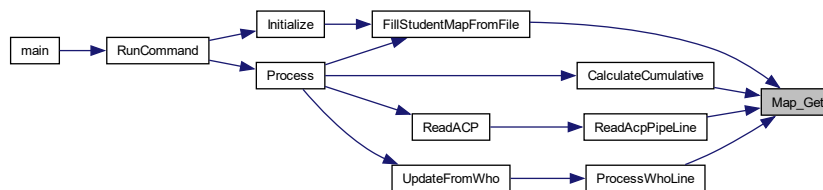
A `map_get_result` containing the sought data.

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 )
  
```

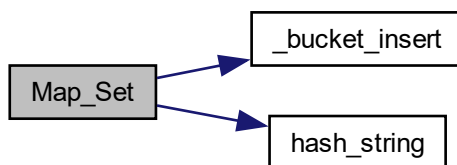
Sets a value in the map.

Parameters

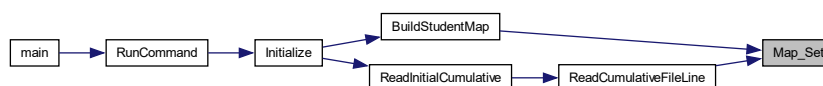
<i>map</i>	The map to set a key in.
<i>key</i>	The key to use.
<i>keylen</i>	The length of the key.
<i>value</i>	The pointer to the data stored at that location.

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

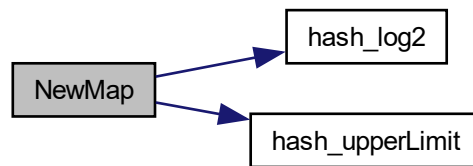
<i>capacity</i>	The estimated required capacity of the map.
-----------------	---

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.](#)

```

00001 #ifndef map_h
00002 #define map_h
00003
00043 // -----
00044 //           Hashing Math
00045 // -----
00046
00052 int hash_log2(int number_to_log);
00053
00062 int hash_string(int hash_table_capacity, char *string, int strlen);
00063
00069 int hash_upperLimit(int bitsize);
00070
00071 // -----
00072 //           General Map Operations
00073 // -----
00074
00080 struct _map_bucket
00081 {
00082     // The key associated with this bucket.
00083     char *key;
00084     // The data this bucket holds.
00085     void *data;
00086     // The next node in this linked list, or NULL if it is a leaf.
00087     struct _map_bucket *next;
00088 };
00089
00099 typedef struct
00100 {
00101     // The number of base buckets in this map.
00102     int size;
00103     // The buckets for this map.
00104     struct _map_bucket *buckets;
00105 } map;
00106
00110 typedef struct
00111 {
  
```

```

00112     // 1 if succesfully found.  0 if not found.
00113     short found;
00114     // The data linked with that key; indeterminate if found == 0.
00115     void *data;
00116 } map_result;
00117
00124 map *NewMap(int capacity);
00125
00133 void Map_Set(map *a_map, char *key, void *value);
00134
00141 map_result Map_Get(map *a_map, char *key);
00142
00150 map_result Map_Delete(map *a_map, char *key, short free_it);
00151
00152 #endif

```

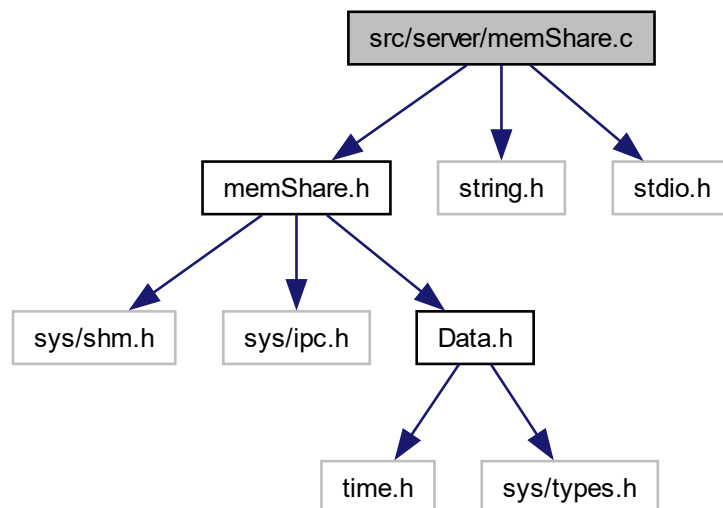
5.19 src/server/memShare.c File Reference

```

#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

<code>shared_mem↔ _id</code>	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 {
```

```

00014     return shmget(MEM_KEY, MEM_SIZE, IPC_CREAT | MEM_PERMISSIONS);
00015 }
00016
00017 int DestroySharedMemory()
00018 {
00019     int shm_id = shmget(MEM_KEY, MEM_SIZE, 0);
00020     int control_result = shmctl(shm_id, IPC_RMID, 0);
00021     if (control_result != -1)
00022         return 0;
00023     return control_result;
00024 }
00025
00026 void *GetMemoryPointer(int shared_mem_id)
00027 {
00028     return shmat(shared_mem_id, NULL, 0);
00029 }
00030
00031 int ReleaseMemoryPointer(void *shmaddr)
00032 {
00033     return shmdt(shmaddr);
00034 }

```

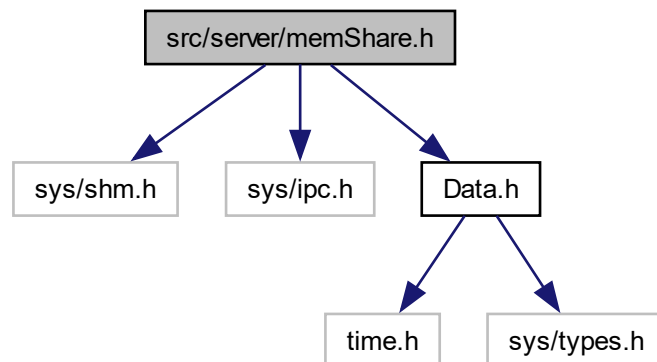
5.21 src/server/memShare.h File Reference

```

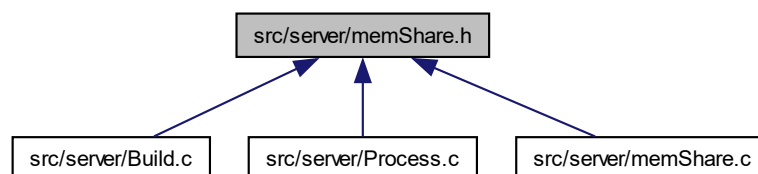
#include <sys/shm.h>
#include <sys/ipc.h>
#include "Data.h"

```

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

<code>shared_mem↔ _id</code>	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.](#)

```
00001 #ifndef MEM_SHARE_H
00002 #define MEM_SHARE_H
00007 #include <sys/shm.h>
00008 #include <sys/ipc.h>
00009 #include "Data.h"
00010
00011 /*
00012 The shared memory key that clients and servers will use to identify the segment.
00013 */
00014 #define MEM_KEY 0x727
00015
```

```

00016 /*
00017 Memory permissions are:
00018 Self:      RW      110 = 6
00019 Group:    R       100 = 4
00020 Others:   R       100 = 4
00021 - All groups can read.
00022 - Self can write.
00023 - None can execute.
00024 */
00025 #define MEM_PERMISSIONS 0664
00026
00027 /*
00028 The memory allocation must as large as the data size times the number of records.
00029 */
00030 #define MEM_SIZE DATA_SIZE *DATA_NUM_RECORDS
00031
00032 int CreateSharedMemory();
00033
00034 int DestroySharedMemory();
00035
00036 void *GetMemoryPointer(int shared_mem_id);
00037
00038 /*
00039 Release a shm memory pointer.
00040 */
00041 int ReleaseMemoryPointer(void *shmaddr);
00042
00043 #endif

```

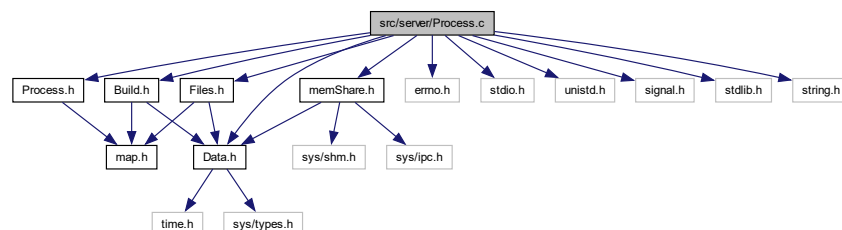
5.23 src/server/Process.c File Reference

```

#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.

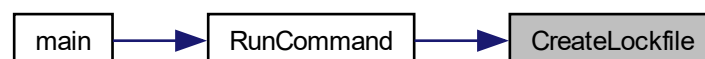
The lockfile will carry a 'data reset' signal and a process ID. CreateLockfile will write the current processes PID.

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 ( )
```

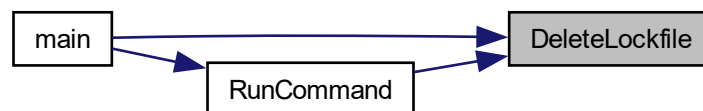
Deletes the lockfile.

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 ( )
```

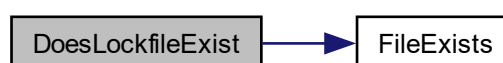
Determines if lockfile exists, which indicates that a server process is already running.

Returns

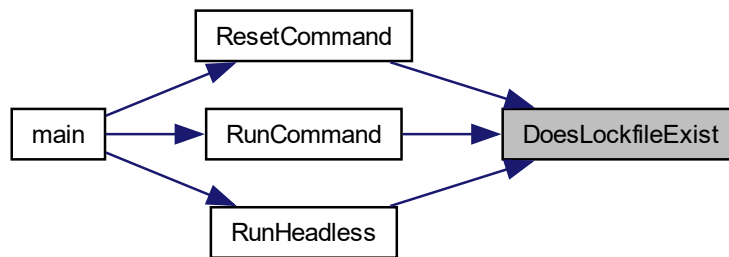
0 if lockfile does not exist, 1 if it does.

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

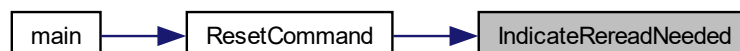
should only be called by non main processes

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

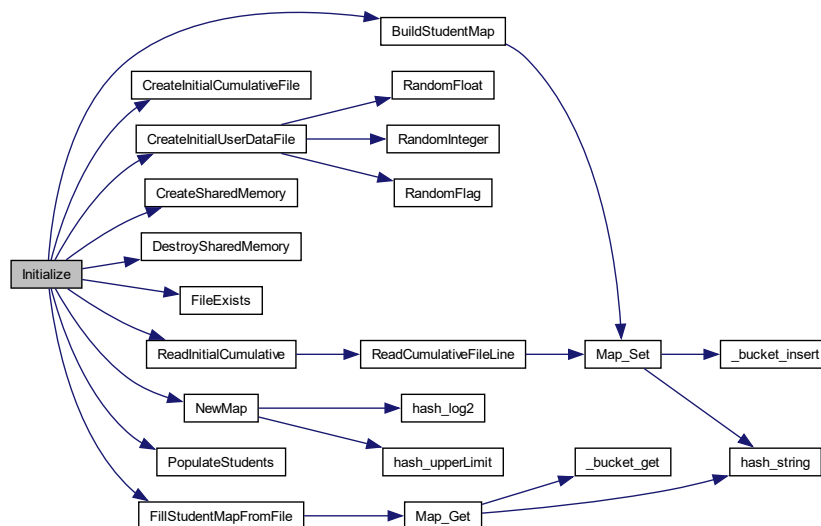
- (1) - Create an initial user data file if it doesn't exist
- (2) - Initialize the students array.
- (3) - Initialize the students map.
- (4) - Read the data from the user data file into the map/array.
- (5) - Initializes the shared memory segment.

Returns

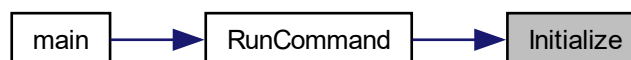
The ID of the shared memory segment. If -1, there was an error.

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

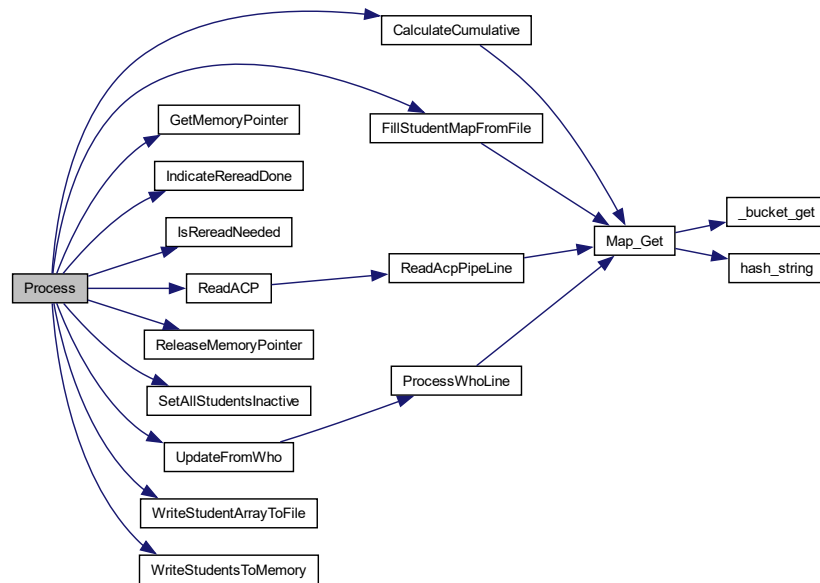
- (1) - Sets all users to inactive.
- (2) - Reads the result of the `who` command, setting some users to active, and possibly changing 'dirty' and last login times.
- (3) - Overwrites the user data file if we are dirty.
- (4) - Sets dirty to false.
- (5) - Re-writes the shared memory.

Parameters

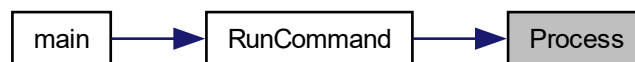
<i>shm_id</i>	The ID of the shared memory segment.
---------------	--------------------------------------

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

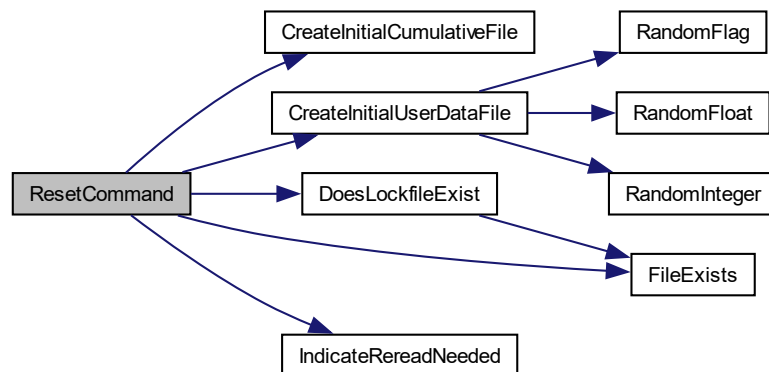
To execute the command, pass "reset" as an argument to the program.

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 ( )
```

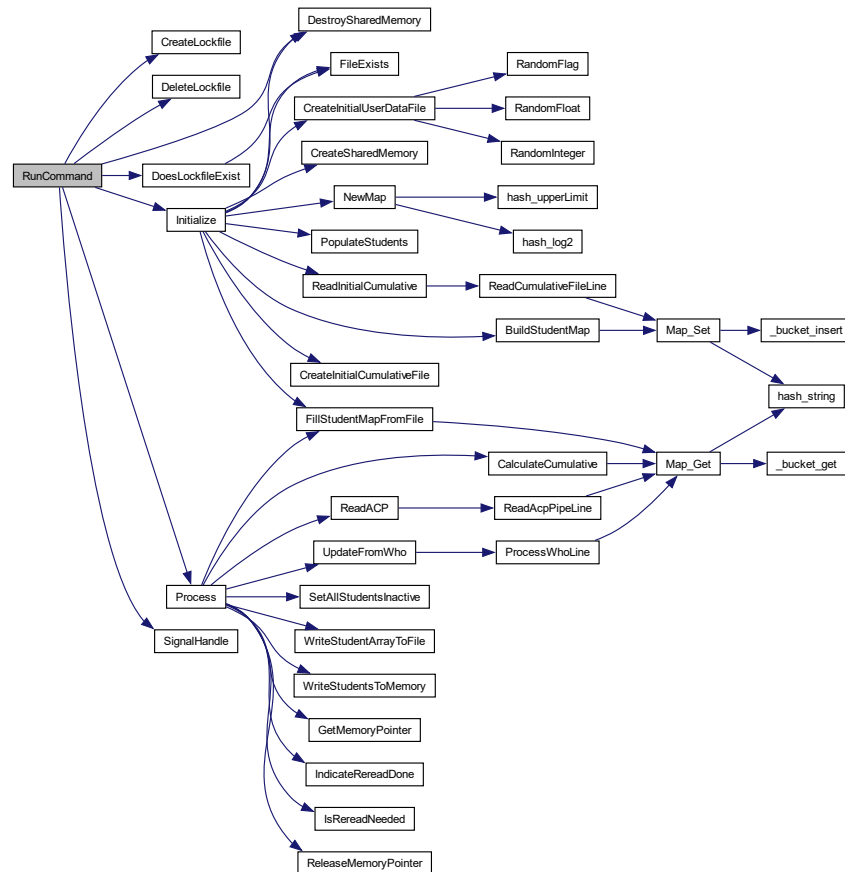
If a server exists, stops it. Begins the process loop.

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 process headlessly.

Parameters

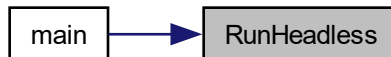
<i>processName</i>	The name of the currently running process.
--------------------	--

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

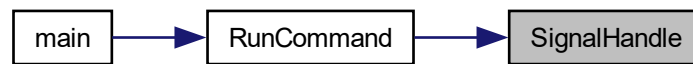
<i>signo</i>	The signal number. Will be <code>SIGTERM</code> from the other server process or <code>SIGINT</code> if interrupted from the console.
--------------	---

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;

```

```

00070     err = fseek(file, 0, SEEK_SET);
00071     if (!err)
00072     {
00073         fputc('1', file);
00074     }
00075     err = fclose(file);
00076     return err;
00077 }
00078
00079 int IndicateRereadDone()
00080 {
00081     FILE *file = fopen(LOCKFILE, "r+");
00082     if (file == NULL)
00083     {
00084         return -1;
00085     }
00086     int err = 0;
00087     err = fseek(file, 0, SEEK_SET);
00088     if (!err)
00089     {
00090         fputc('0', file);
00091     }
00092     err = fclose(file);
00093     return err;
00094 }
00095
00096 short IsRereadNeeded()
00097 {
00098     FILE *file = fopen(LOCKFILE, "r");
00099     char firstc = fgetc(file);
00100     fclose(file);
00101     return firstc == '1';
00102 }
00103
00104 void SignalHandle(int signo)
00105 {
00106     printf("Received shutdown signal.\n");
00107     if (signo == SIGINT || signo == SIGTERM)
00108     {
00109         is_stopping = 1;
00110     }
00111 }
00112
00113 short is_stopping = 0;
00114
00115 // ~~~~~ CLI Commands ~~~~~
00116
00117 int Initialize()
00118 {
00119     int err;
00120     if (!FileExists(STATIC_USER_DATA_FILE))
00121     {
00122         printf("%s does not exist. Creating.\n", STATIC_USER_DATA_FILE);
00123         err = CreateInitialUserDataFile(STATIC_USER_DATA_FILE, Data_IDs, DATA_NUM_RECORDS);
00124         if (err)
00125         {
00126             printf("Problem creating %s!\n", STATIC_USER_DATA_FILE);
00127         }
00128     }
00129     if (!FileExists(STATIC_USER_CUMULATIVE_FILE))
00130     {
00131         printf("%s does not exist. Creating.\n", STATIC_USER_CUMULATIVE_FILE);
00132         err = CreateInitialCumulativeFile(STATIC_USER_CUMULATIVE_FILE);
00133         if (err)
00134         {
00135             printf("Problem creating %s!\n", STATIC_USER_CUMULATIVE_FILE);
00136         }
00137     }
00138     PopulateStudents(Data_IDs, Data_Names, DATA_NUM_RECORDS);
00139     student_map = NewMap(50);
00140     BuildStudentMap(student_map, students, DATA_NUM_RECORDS);
00141     err = FillStudentMapFromFile(student_map, STATIC_USER_DATA_FILE, Data_IDs, DATA_NUM_RECORDS);
00142     if (err)
00143     {
00144         printf("Problem filling student map from %s!\n", STATIC_USER_DATA_FILE);
00145     }
00146     printf("Student data retrieved from file.\n");
00147     initial_cumulative_times = NewMap(50);
00148     err = ReadInitialCumulative(initial_cumulative_times, STATIC_USER_CUMULATIVE_FILE);
00149     if (err)
00150     {
00151         printf("Failed to read %s. Cumulative times may be wrong!", STATIC_USER_CUMULATIVE_FILE);
00152     }
00153 }
00154
00155 dirty = 0;
00156

```



```

00157     int shm_id = CreateSharedMemory();
00158     if (shm_id == -1)
00159     {
00160         DestroySharedMemory();
00161         shm_id = CreateSharedMemory();
00162     }
00163     printf("Shared memory allocated.\n");
00164     return shm_id;
00165 }
00166
00167 void Process(int shm_id)
00168 {
00169     if (IsRereadNeeded())
00170     {
00171         printf("\nReread indicated - rechecking user data file.");
00172         FillStudentMapFromFile(student_map, STATIC_USER_DATA_FILE, Data_IDs, DATA_NUM_RECORDS);
00173         IndicateRereadDone();
00174     }
00175     SetAllStudentsInactive(students, DATA_NUM_RECORDS);
00176     int err = ReadACP(student_map);
00177     if (err)
00178     {
00179         printf("Error piping ac -p command! \n");
00180     }
00181     else
00182     {
00183         CalculateCumulative(students, DATA_NUM_RECORDS, initial_cumulative_times);
00184     }
00185     err = UpdateFromWho(student_map);
00186     if (err)
00187     {
00188         perror("Error updating from who!");
00189     }
00190     if (dirty)
00191     {
00192         err = WriteStudentArrayToFile(students, DATA_NUM_RECORDS, STATIC_USER_DATA_FILE);
00193         if (err)
00194         {
00195             printf("\nError updating %s!", STATIC_USER_DATA_FILE);
00196         }
00197         else
00198         {
00199             dirty = 0;
00200         }
00201     }
00202     void *ptr = GetMemoryPointer(shm_id);
00203     if (ptr == (void *)-1)
00204     {
00205         perror("Error attaching to shared memory");
00206     }
00207     else
00208     {
00209         WriteStudentsToMemory(ptr, students, DATA_NUM_RECORDS);
00210         ReleaseMemoryPointer(ptr);
00211     }
00212 }
00213
00214 void HelpCommand()
00215 {
00216     printf("\nUsage:  server [OPTION]\n\n");
00217     printf("Options:  \n");
00218     printf("\thelp\t\t\tShows the possible program commands\n");
00219     printf("\treset\t\t\tRegenerates the user data file\n");
00220     printf("\tstop\t\t\tStops an existing server process if it is running\n");
00221     printf("\trun\t\t\tCreates a new server with output to the shell if a server isn't already\n");
00222     printf("\theadless\t\tCreates a new headless server if a server isn't already running.\n\n");
00223 }
00224
00225 void RunCommand()
00226 {
00227     printf("\nRunning server.\n");
00228     if (DoesLockfileExist())
00229     {
00230         printf("\nServer is already running.  Run 'server stop' to shut it down first.\n");
00231         return;
00232     }
00233     int err = CreateLockfile();
00234     if (err)
00235     {
00236         printf("\nFailed to create lockfile!  Exiting.\n");
00237         return;
00238     }
00239     int shm_id = Initialize();
00240     signal(SIGTERM, SignalHandle);
00241     signal(SIGINT, SignalHandle);
00242     printf("Server started.\n");

```

```

00243     fflush(stdout);
00244     while (!is_stopping)
00245     {
00246         Process(shm_id);
00247         sleep(1);
00248     }
00249     printf("Server shutting down.\n");
00250     err = DeleteLockfile();
00251     if (err)
00252     {
00253         printf("Failed to delete lockfile!\n");
00254     }
00255     err = DestroySharedMemory();
00256     if (err)
00257     {
00258         printf("Failed to destroy shared memory!\n");
00259     }
00260     printf("Server terminated.\n");
00261 }
00262
00263 void StopCommand()
00264 {
00265     printf("\nStopping server...\n");
00266     int err = TerminateExistingServer();
00267     if (err)
00268     {
00269         if (err == -1)
00270         {
00271             printf("Server isn't running.\n");
00272         }
00273         else if (err == -2)
00274         {
00275             printf("Lockfile did not contain a valid process id!\n");
00276         }
00277         else
00278         {
00279             printf("Sending terminate signal failed!\n");
00280         }
00281     }
00282     else
00283     {
00284         printf("Server terminated.\n");
00285     }
00286 }
00287
00288 void ResetCommand()
00289 {
00290     int err;
00291
00292     if (FileExists(STATIC_USER_DATA_FILE))
00293     {
00294         printf("User data file exists. Deleting...\n");
00295         remove(STATIC_USER_DATA_FILE);
00296     }
00297
00298     printf("Creating new data file.\n");
00299     err = CreateInitialUserDataFile(STATIC_USER_DATA_FILE, Data_IDs, DATA_NUM_RECORDS);
00300     if (err)
00301     {
00302         printf("Problem creating %s!\n", STATIC_USER_DATA_FILE);
00303     }
00304     else
00305     {
00306         printf("%s created.\n", STATIC_USER_DATA_FILE);
00307     }
00308
00309     printf("Creating new cumulative file.\n");
00310     err = CreateInitialCumulativeFile(STATIC_USER_CUMULATIVE_FILE);
00311     if (err)
00312     {
00313         printf("Problem creating %s!\n", STATIC_USER_CUMULATIVE_FILE);
00314     }
00315     else
00316     {
00317         printf("%s created.\n", STATIC_USER_CUMULATIVE_FILE);
00318     }
00319
00320     if (DoesLockfileExist())
00321     {
00322         printf("Indicated re-read to running server process.\n");
00323         IndicateRereadNeeded();
00324     }
00325 }
00326
00327 void RunHeadless(char *processName)
00328 {
00329     if (DoesLockfileExist())

```

```

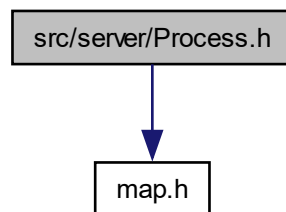
00330     {
00331         printf("Server process already running.\n");
00332         return;
00333     }
00334     char commandFront[] = " nohup ";
00335     char commandEnd[] = " run & exit";
00336     size_t comm_length = strlen(commandFront) + strlen(commandEnd) + strlen(processName) + 1;
00337     char *commandFull = malloc(comm_length * sizeof(char));
00338     memset(commandFull, 0, comm_length * sizeof(char));
00339     strcpy(commandFull, commandFront);
00340     strcat(commandFull, processName);
00341     strcat(commandFull, commandEnd);
00342
00343     printf("Executing: %s\n", commandFull);
00344     popen(commandFull, "we");
00345     printf("Server running headlessly.\n");
00346 }

```

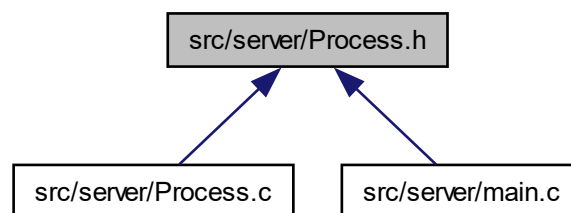
5.25 src/server/Process.h File Reference

```
#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.

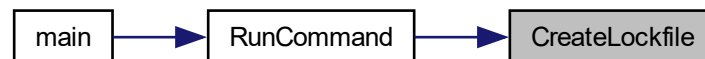
The lockfile will carry a 'data reset' signal and a process ID. CreateLockfile will write the current processes PID.

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 ( )
```

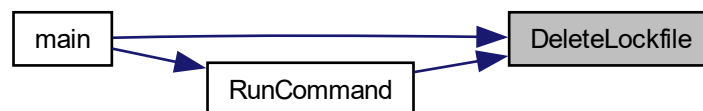
Deletes the lockfile.

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

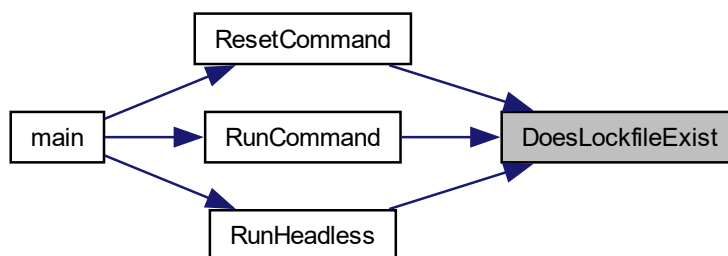
0 if lockfile does not exist, 1 if it does.

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

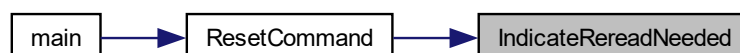
should only be called by non main processes

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

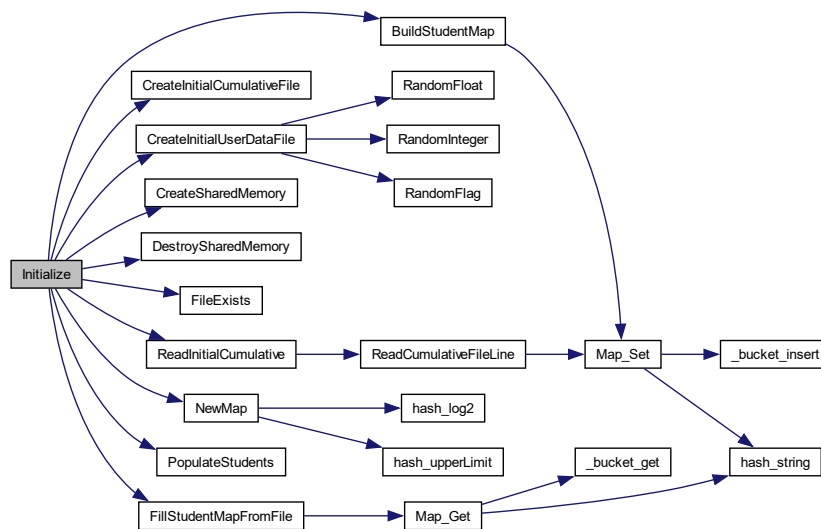
- (1) - Create an initial user data file if it doesn't exist
- (2) - Initialize the students array.
- (3) - Initialize the students map.
- (4) - Read the data from the user data file into the map/array.
- (5) - Initializes the shared memory segment.

Returns

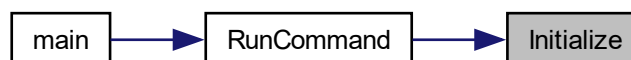
The ID of the shared memory segment. If -1, there was an error.

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

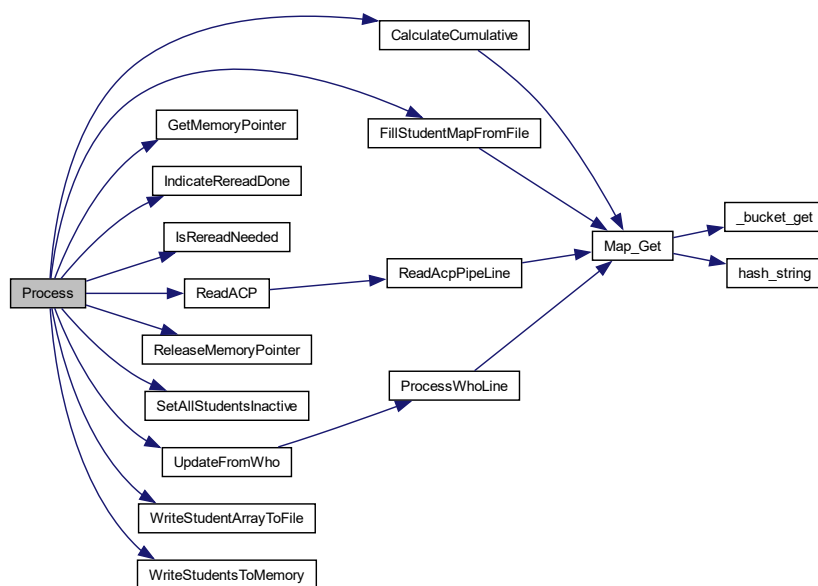
- (1) - Sets all users to inactive.
- (2) - Reads the result of the `who` command, setting some users to active, and possibly changing 'dirty' and last login times.
- (3) - Overwrites the user data file if we are dirty.
- (4) - Sets dirty to false.
- (5) - Re-writes the shared memory.

Parameters

<i>shm_id</i>	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

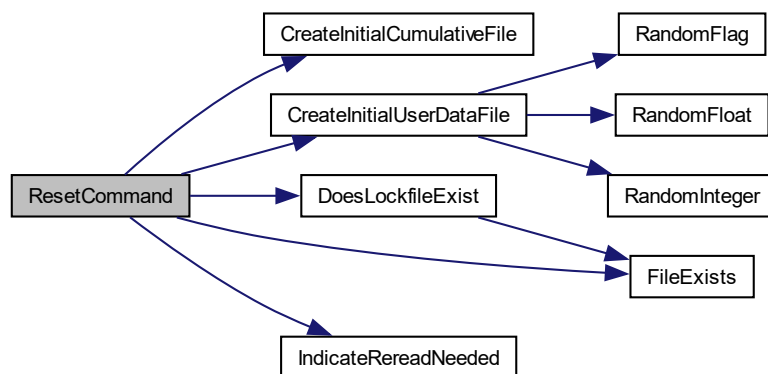
To execute the command, pass "reset" as an argument to the program.

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 ( )
```

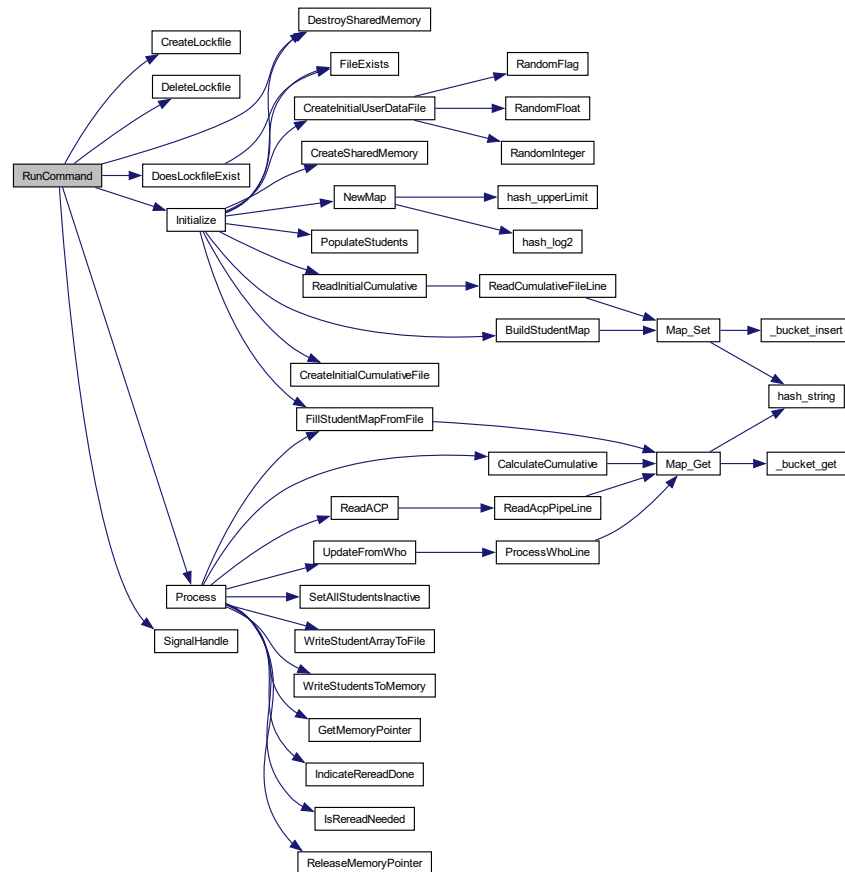
If a server exists, stops it. Begins the process loop.

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 )
```

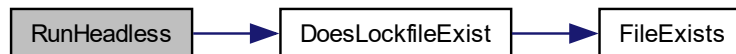
Uses `nohup ./processName` to run the process headlessly.

Parameters

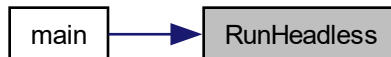
<i>processName</i>	The name of the currently running process.
--------------------	--

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

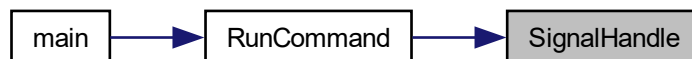
<i>signo</i>	The signal number. Will be <code>SIGTERM</code> from the other server process or <code>SIGINT</code> if interrupted from the console.
--------------	---

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.](#)

```

00001 #ifndef Process_h
00002 #define Process_h
00007 #include "map.h"
00008
00017 #define LOCKFILE "/tmp/ecet-server.lock"
00018
00019 // ~~~~~ Lockfile Commands ~~~~~
00020
00026 short DoesLockfileExist();
00027
00036 int CreateLockfile();
00037
00042 int DeleteLockfile();
00043
00052 int TerminateExistingServer();
00053
00061 int IndicateRereadNeeded();
00062
00068 int IndicateRereadDone();
00069
00075 short IsRereadNeeded();
00076
00084 void SignalHandle(int signo);
00085
00089 extern short is_stopping;
00090
00091 // ~~~~~ CLI Commands ~~~~~
00092
00093 extern map *student_map;
00094
00105 int Initialize();
00106
00119 void Process(int shm_id);
00120
00126 void ClearCommand();
00127
00134 void ResetCommand();
00135
00141 void StopCommand();
00142
00148 void RunCommand();
00149
00156 void HelpCommand();
00157
00162 void RunHeadless(char *processName);
00163
00164 #endif

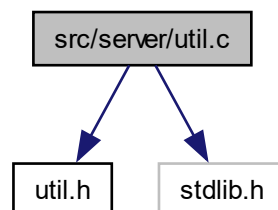
```

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

<i>percentage_chance</i>	The chance to return 1.
--------------------------	-------------------------

Note

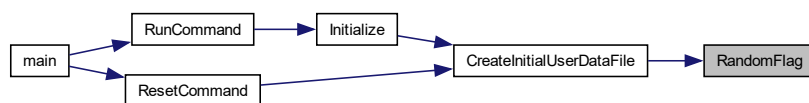
If percentage_chance > 1, this will always return true.

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

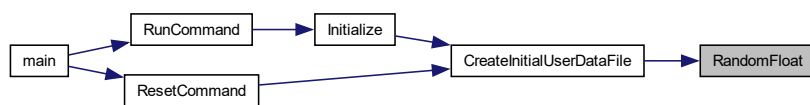
<i>min</i>	The minimum, inclusive.
<i>max</i>	The maximum, inclusive.

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 )

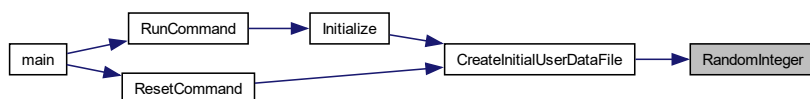
```

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.](#)

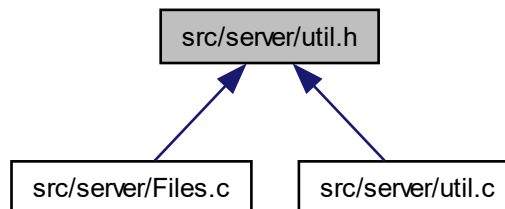
```

00001
00004 #include "util.h"
00005
00006 #include <stdlib.h>
00007
00008 int RandomInteger(int min, int max)
00009 {
00010     int r_add = rand() % (max - min + 1);
00011     return r_add + min;
00012 }
00013
00014 float RandomFloat(float min, float max)
00015 {
00016     float dif = max - min;
00017     int rand_int = rand() % (int)(dif * 10000);
00018     return min + (float)rand_int / 10000.0;
00019 }
00020
00021 short RandomFlag(float percentage_chance)
00022 {
00023     float random_value = (float)rand() / RAND_MAX;
00024     if (random_value < percentage_chance)
00025     {
00026         return 1;
00027     }
00028     return 0;
00029 }

```

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

<i>percentage_chance</i>	The chance to return 1.
--------------------------	-------------------------

Note

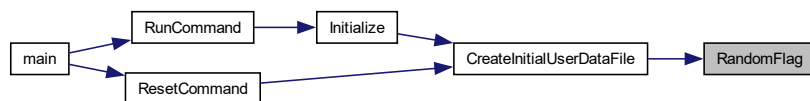
If `percentage_chance > 1`, this will always return true.

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

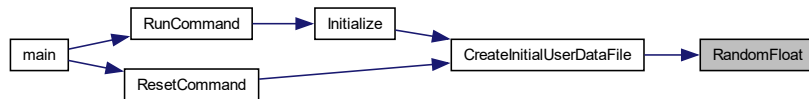
<i>min</i>	The minimum, inclusive.
<i>max</i>	The maximum, inclusive.

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

<i>min</i>	The minimum, inclusive.
<i>max</i>	The maximum, inclusive.

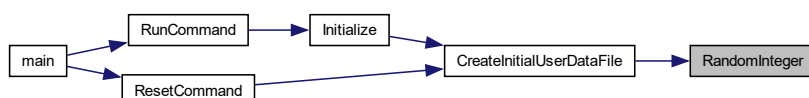
Returns

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.](#)

```
00001 #ifndef util_h
00002 #define util_h
00015 int RandomInteger(int min, int max);
00016
00023 float RandomFloat(float min, float max);
00024
00031 short RandomFlag(float percentage_chance);
00032
00033 #endif
```


Index

- [_bucket_delete](#)
 - [map.c, 57](#)
 - [_bucket_get](#)
 - [map.c, 57](#)
 - [_bucket_insert](#)
 - [map.c, 58](#)
- [active](#)
 - [Student, 9](#)
- [age](#)
 - [Student, 10](#)
- [buckets](#)
 - [map, 8](#)
- [Build.c](#)
 - [BuildStudentMap, 12](#)
 - [CalculateCumulative, 13](#)
 - [dirty, 21](#)
 - [PopulateStudents, 13](#)
 - [ProcessWhoLine, 14](#)
 - [ReadACP, 15](#)
 - [ReadAcpPipeLine, 16](#)
 - [ReadCumulativeFileLine, 17](#)
 - [ReadInitialCumulative, 18](#)
 - [SetAllStudentsInactive, 19](#)
 - [students, 21](#)
 - [UpdateFromWho, 19](#)
 - [WriteStudentsToMemory, 20](#)
- [Build.h](#)
 - [BuildStudentMap, 26](#)
 - [CalculateCumulative, 27](#)
 - [dirty, 35](#)
 - [PopulateStudents, 27](#)
 - [ProcessWhoLine, 28](#)
 - [ReadACP, 29](#)
 - [ReadAcpPipeLine, 30](#)
 - [ReadCumulativeFileLine, 31](#)
 - [ReadInitialCumulative, 32](#)
 - [SetAllStudentsInactive, 33](#)
 - [students, 35](#)
 - [UpdateFromWho, 33](#)
 - [WriteStudentsToMemory, 34](#)
- [BuildStudentMap](#)
 - [Build.c, 12](#)
 - [Build.h, 26](#)
- [CalculateCumulative](#)
 - [Build.c, 13](#)
 - [Build.h, 27](#)
- [char_ratio](#)
 - [map.c, 64](#)
- [ClearCommand](#)
 - [Process.h, 100](#)
- [CreateInitialCumulativeFile](#)
 - [Files.c, 42](#)
 - [Files.h, 49](#)
- [CreateInitialUserDataFile](#)
 - [Files.c, 42](#)
 - [Files.h, 50](#)
- [CreateLockfile](#)
 - [Process.c, 83](#)
 - [Process.h, 101](#)
- [CreateSharedMemory](#)
 - [memShare.c, 75](#)
 - [memShare.h, 79](#)
- [data](#)
 - [map_result, 8](#)
- [Data.c](#)
 - [Data_IDs, 37](#)
 - [Data_Names, 37](#)
- [Data.h](#)
 - [DATA_ID_MAX_LENGTH, 39](#)
 - [Data_IDs, 40](#)
 - [DATA_NAME_MAX_LENGTH, 39](#)
 - [Data_Names, 40](#)
 - [DATA_NUM_RECORDS, 40](#)
 - [DATA_SIZE, 40](#)
- [DATA_ID_MAX_LENGTH](#)
 - [Data.h, 39](#)
- [Data_IDs](#)
 - [Data.c, 37](#)
 - [Data.h, 40](#)
- [DATA_NAME_MAX_LENGTH](#)
 - [Data.h, 39](#)
- [Data_Names](#)
 - [Data.c, 37](#)
 - [Data.h, 40](#)
- [DATA_NUM_RECORDS](#)
 - [Data.h, 40](#)
- [DATA_SIZE](#)
 - [Data.h, 40](#)
- [DeleteLockfile](#)
 - [Process.c, 83](#)
 - [Process.h, 101](#)
- [DestroySharedMemory](#)
 - [memShare.c, 76](#)
 - [memShare.h, 80](#)
- [dirty](#)
 - [Build.c, 21](#)

- Build.h, 35
- DoesLockfileExist
 - Process.c, 84
 - Process.h, 102
- FileExists
 - Files.c, 43
 - Files.h, 51
- Files.c
 - CreateInitialCumulativeFile, 42
 - CreateInitialUserDataFile, 42
 - FileExists, 43
 - FillStudentMapFromFile, 44
 - WriteStudentArrayToFile, 45
- Files.h
 - CreateInitialCumulativeFile, 49
 - CreateInitialUserDataFile, 50
 - FileExists, 51
 - FillStudentMapFromFile, 52
 - STATIC_USER_CUMULATIVE_FILE, 49
 - STATIC_USER_DATA_FILE, 49
 - WriteStudentArrayToFile, 53
- FillStudentMapFromFile
 - Files.c, 44
 - Files.h, 52
- found
 - map_result, 9
- fullName
 - Student, 10
- GetMemoryPointer
 - memShare.c, 76
 - memShare.h, 80
- gpa
 - Student, 10
- hash_log2
 - map.c, 58
 - map.h, 67
- hash_string
 - map.c, 59
 - map.h, 69
- hash_upperLimit
 - map.c, 60
 - map.h, 70
- HelpCommand
 - Process.c, 85
 - Process.h, 103
- IndicateRereadDone
 - Process.c, 85
 - Process.h, 103
- IndicateRereadNeeded
 - Process.c, 86
 - Process.h, 104
- initial_cumulative_times
 - Process.c, 94
- Initialize
 - Process.c, 86
- Process.h, 104
- is_stopping
 - Process.c, 94
 - Process.h, 112
- IsRereadNeeded
 - Process.c, 87
 - Process.h, 105
- lastLogin
 - Student, 10
- LOCKFILE
 - Process.h, 100
- loginDuration
 - Student, 10
- main
 - main.c, 54
- main.c
 - main, 54
- map, 7
 - buckets, 8
 - size, 8
- map.c
 - _bucket_delete, 57
 - _bucket_get, 57
 - _bucket_insert, 58
 - char_ratio, 64
 - hash_log2, 58
 - hash_string, 59
 - hash_upperLimit, 60
 - Map_Delete, 60
 - Map_Get, 61
 - Map_Set, 62
 - NewMap, 63
- map.h
 - hash_log2, 67
 - hash_string, 69
 - hash_upperLimit, 70
 - Map_Delete, 70
 - Map_Get, 71
 - Map_Set, 72
 - NewMap, 73
- Map_Delete
 - map.c, 60
 - map.h, 70
- Map_Get
 - map.c, 61
 - map.h, 71
- map_result, 8
 - data, 8
 - found, 9
- Map_Set
 - map.c, 62
 - map.h, 72
- MEM_KEY
 - memShare.h, 79
- MEM_PERMISSIONS
 - memShare.h, 79
- MEM_SIZE

- memShare.h, 79
- memShare.c
 - CreateSharedMemory, 75
 - DestroySharedMemory, 76
 - GetMemoryPointer, 76
 - ReleaseMemoryPointer, 77
- memShare.h
 - CreateSharedMemory, 79
 - DestroySharedMemory, 80
 - GetMemoryPointer, 80
 - MEM_KEY, 79
 - MEM_PERMISSIONS, 79
 - MEM_SIZE, 79
 - ReleaseMemoryPointer, 81
- NewMap
 - map.c, 63
 - map.h, 73
- PopulateStudents
 - Build.c, 13
 - Build.h, 27
- Process
 - Process.c, 88
 - Process.h, 106
- Process.c
 - CreateLockfile, 83
 - DeleteLockfile, 83
 - DoesLockfileExist, 84
 - HelpCommand, 85
 - IndicateRereadDone, 85
 - IndicateRereadNeeded, 86
 - initial_cumulative_times, 94
 - Initialize, 86
 - is_stopping, 94
 - IsRereadNeeded, 87
 - Process, 88
 - ResetCommand, 89
 - RunCommand, 90
 - RunHeadless, 91
 - SignalHandle, 92
 - StopCommand, 93
 - student_map, 94
 - TerminateExistingServer, 93
- Process.h
 - ClearCommand, 100
 - CreateLockfile, 101
 - DeleteLockfile, 101
 - DoesLockfileExist, 102
 - HelpCommand, 103
 - IndicateRereadDone, 103
 - IndicateRereadNeeded, 104
 - Initialize, 104
 - is_stopping, 112
 - IsRereadNeeded, 105
 - LOCKFILE, 100
 - Process, 106
 - ResetCommand, 107
 - RunCommand, 108
- RunHeadless, 109
- SignalHandle, 110
- StopCommand, 111
- student_map, 112
- TerminateExistingServer, 111
- ProcessWhoLine
 - Build.c, 14
 - Build.h, 28
- RandomFlag
 - util.c, 114
 - util.h, 116
- RandomFloat
 - util.c, 114
 - util.h, 117
- RandomInteger
 - util.c, 115
 - util.h, 118
- ReadACP
 - Build.c, 15
 - Build.h, 29
- ReadAcpPipeLine
 - Build.c, 16
 - Build.h, 30
- ReadCumulativeFileLine
 - Build.c, 17
 - Build.h, 31
- ReadInitialCumulative
 - Build.c, 18
 - Build.h, 32
- ReleaseMemoryPointer
 - memShare.c, 77
 - memShare.h, 81
- ResetCommand
 - Process.c, 89
 - Process.h, 107
- RunCommand
 - Process.c, 90
 - Process.h, 108
- RunHeadless
 - Process.c, 91
 - Process.h, 109
- SetAllStudentsInactive
 - Build.c, 19
 - Build.h, 33
- SignalHandle
 - Process.c, 92
 - Process.h, 110
- size
 - map, 8
- src/server/Build.c, 11, 22
- src/server/Build.h, 24, 36
- src/server/Data.c, 36, 38
- src/server/Data.h, 38, 41
- src/server/Files.c, 41, 46
- src/server/Files.h, 47, 53
- src/server/main.c, 54, 55
- src/server/map.c, 56, 64

- src/server/map.h, [67](#), [74](#)
- src/server/memShare.c, [75](#), [77](#)
- src/server/memShare.h, [78](#), [81](#)
- src/server/Process.c, [82](#), [95](#)
- src/server/Process.h, [99](#), [113](#)
- src/server/util.c, [113](#), [116](#)
- src/server/util.h, [116](#), [119](#)
- STATIC_USER_CUMULATIVE_FILE
 - Files.h, [49](#)
- STATIC_USER_DATA_FILE
 - Files.h, [49](#)
- StopCommand
 - Process.c, [93](#)
 - Process.h, [111](#)
- Student, [9](#)
 - active, [9](#)
 - age, [10](#)
 - fullName, [10](#)
 - gpa, [10](#)
 - lastLogin, [10](#)
 - loginDuration, [10](#)
 - userID, [10](#)
- student_map
 - Process.c, [94](#)
 - Process.h, [112](#)
- students
 - Build.c, [21](#)
 - Build.h, [35](#)
- TerminateExistingServer
 - Process.c, [93](#)
 - Process.h, [111](#)
- UpdateFromWho
 - Build.c, [19](#)
 - Build.h, [33](#)
- userID
 - Student, [10](#)
- util.c
 - RandomFlag, [114](#)
 - RandomFloat, [114](#)
 - RandomInteger, [115](#)
- util.h
 - RandomFlag, [116](#)
 - RandomFloat, [117](#)
 - RandomInteger, [118](#)
- WriteStudentArrayToFile
 - Files.c, [45](#)
 - Files.h, [53](#)
- WriteStudentsToMemory
 - Build.c, [20](#)
 - Build.h, [34](#)