北京大学 PEKING UNIVERSITY

瀚高 HIGH GO

密钥管理实践

开源开发实践-第六周

David & Cary

CONTENT
目 录

Introduction

Introduction

What To Do Now?

- So far, we have learned the fundamentals and basics of a Key Management System (KMS).
- And the benefits of external KMS over internal.
- Now, it is time to start implementing an internal KMS as a separate module in PostgreSQL.
- We will walk through the Source structure of PostgreSQL and learn how you can add a new component to the system
- We will also discuss about some of the considerations in designing a KMS. What is it supposed to do?

PostgreSQL Source Structure

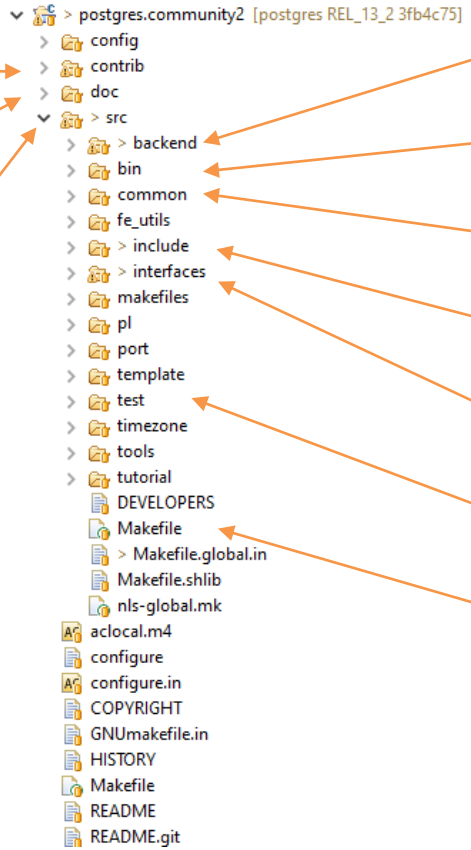
PostgreSQL Root Source Structure

The Root Source Structure

"contrib" folder contains the optional extensions that can be "added" onto PG. Will talk more in later class.

"doc" folder contains all the documentations of PG system written in sgml

"src" folder contains all the core components of PG. Both front end, backend, examples, tests, tools…etc



"backend" sub-folder contains all the backend components that drive PG.

"bin" sub-folder contains front end applications and tools that interacts with PG

"common" sub-folder contains common routines that everyone shares

"include" sub-folder contains all the header files for all the backend modules.

"interfaces" sub-folder contains interface-related or communication-related routines

"test" sub-folder contains all the regression test cases for PG. Will talk more in later class.

"Makefile" defines what to do after you issue the "make" command. Will discuss later

PostgreSQL Frontend Source Structure

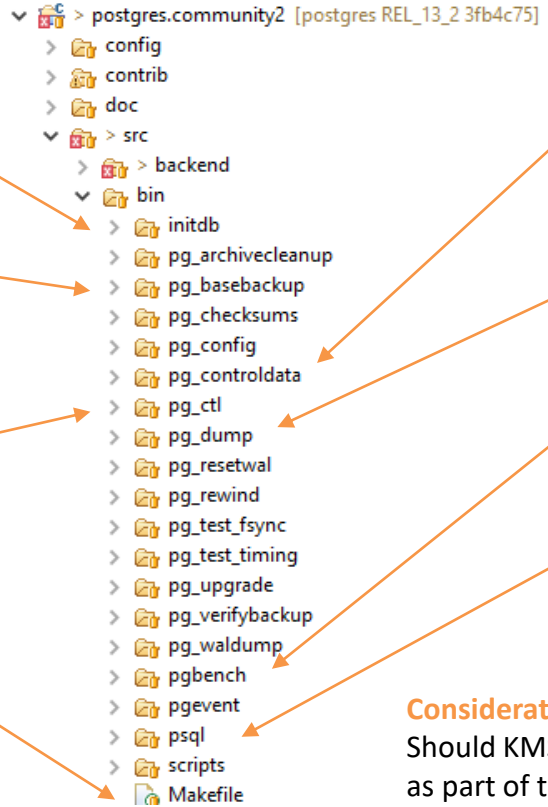
The Frontend Source Structure

"initdb" folder contains initdb binary that initializes a new PG database cluster. This action is also called "bootstrapping"

"pg_basebackup" folder contains pg_basebackup binary that is responsible for taking a physical backup of your database

"pg_ctl" folder contains pg_ctl binary that allows you to start, stop, and restart a PG database

"Makefile" tells the "make" command to go into all the above folder and build each front end binaries



"pg_controldata" folder contains the pg_controldata binary that allows you to check the current operating status of PG database

"pg_dump" folder contains the pg_dump binary that allows you to dump the data of the PG database.

"pgbench" folder contains the pgbench utility that allows you to simulate traffic load to test PG database performance

"psql" folder contains the psql utility that allows you to interacts with a PG database

Consideration:
Should KMS be added here as part of the front end?

PostgreSQL Backend Source Structure

The Backend Source Structure

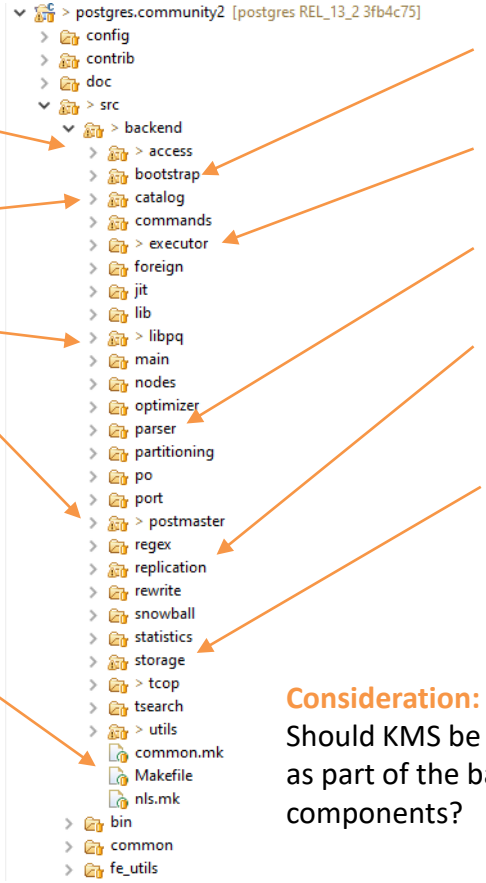
"access" folder contains the access information for tuple data and indexes. Ex: heap, btree index, gist…etc

"catalog" folder contains the logics to interact with catalog tables

"libpq" folder contains the routines for communication between client and server

"postmaster" folder contains the main code for postmaster and how it manages child processes

"Makefile" tells the "make" command to compile and link all the backend components and produce the final output object



"bootstrap" folder contains the initialization routines for PG. Such as during "initdb" process

"executor" folder contains logics to actually execute a "query plan"

"parser" folder contains logics to parse the SQL commands user enters.

"replication" folder contains logics to perform data replication between multiple PG instances

"storage" folder contains logics to handle data read and write between shared memory and the disk. This is the home of "buffer manager"

Consideration:
Should KMS be added here as part of the backend components?

The Makefile

What To Do Now?

- You may have notice that there is a **Makefile** in both the **backend** and the **bin** folder and they behave very differently.
- In fact, there is almost always a Makefile in all folders and sub-folders under the **src directory**.
- So, what exactly is a Makefile?
- A Makefile is basically a series of instructions that tells the "**make**" command what to compile, how to compile, where to compile, how to install, which compiler to use, what flags to use…etc
- This file can get very, very complex! And one Makefile can include other Makefiles!
- This file is very important, especially if you want to add a new components inside PostgreSQL. You will have to have your own Makefile!
- Fortunately, you do not need to create a Makefile from scratch, you can copy the Makefile from another process and change it up a little!
- Before I talk a little more about Makefile, let me show you how compilation works

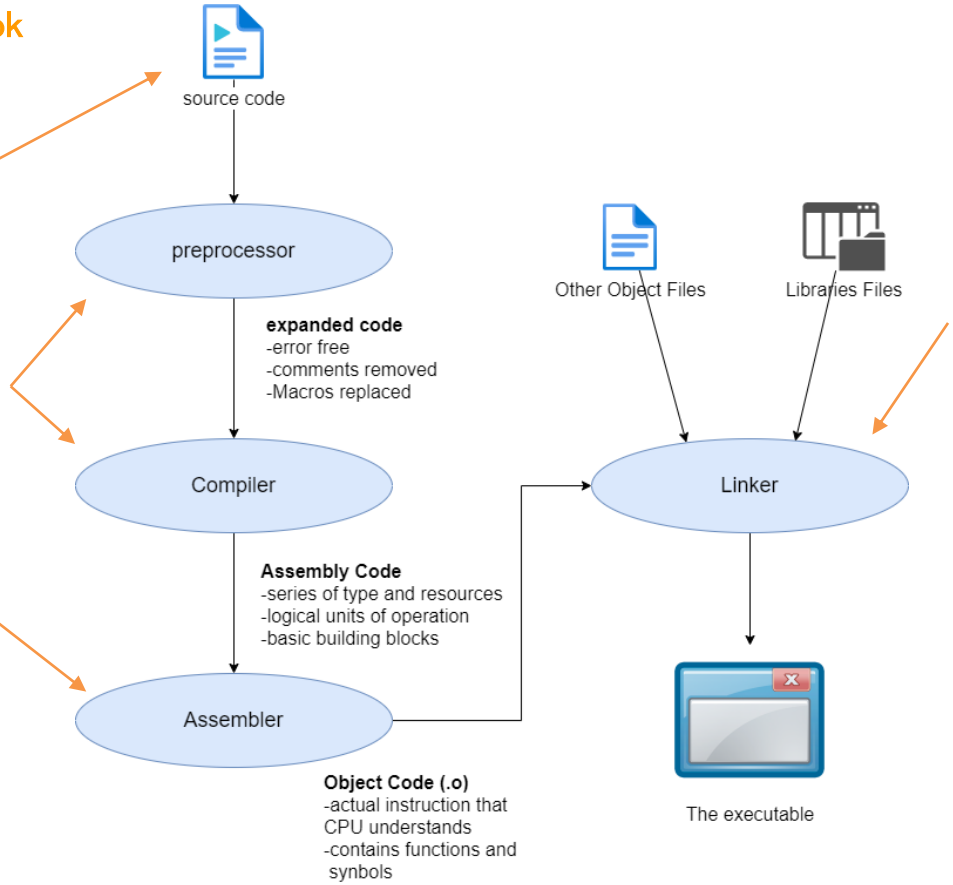
The Compilation Process

A Behind the Scene Look

That is the source file you will be working on. Normally ended with (.c) extension
Ex: bufmgr.c

Preprocessor then checks the syntax and compiler converts the expanded code into an assembly code

Then the assembly converts the assembly code into an object file. Normally every single (.c) file will produce one (.o) file.
Ex: bufmgr.o



Finally, the linker combines all of your object files (.o) including:

- Ones you compiled
- Ones you introduced to the system
- From shared and static libraries (.a) or (.so) files

Into one executable file. This is the final result of the compilation process.

Makefile

Makefile Basics

- The Makefile is a very big topic, and if you would like to learn more about Makefile, you can refer to this link: (https://makefiletutorial.com/)
- A Makefile consists of set of **targets** that you have to define what to do. A target name is set by the colon character (:)
- After the (:) , is the prerequisites, or requirements for this target. Normally the command is a series of Bash scripts and compiler commands

```
targets: prerequisites
    command
    command
    command
```

- Common variables that we use in Makefiles

Variable	Meaning
CC	The compiler to use (ex: gcc)
CFLAGS	The compiler flags to use (ex: -g –O0)
OBJS	List of object files required to build (ex: bufmgr.o buf_init.o freelist.o)
LDFLAGS	Additional paths to locate additional libraries during linking process. (ex: -L/usr/local/lib/ -L/usr/lib)
LIBS	Actual libraries that we want to link. (ex: -lcrypto –lssl)
INCLUDE	Additional paths to locate the header files during compilation process. (ex: -I/usr/local/include –I/usr/include)

Makefile Example

Basic Makefile Example

```
all: hg_smgrserver hg_smgrserver_test

hg_smgrserver: hg_smgrserver.o  hg_netio_utils.o
	$(CC) $(CFLAGS) $(OBJS) $(LDFLAGS) $(LDFLAGS_EX) $(LIBS) -o hg_smgrserver

hg_smgrserver_test: hg_smgrclient_test.o
	$(CC) $(CFLAGS) hg_smgrclient_test.o $(LDFLAGS) $(LDFLAGS_EX) $(LIBS) -o hg_smgrclient_test

install: all
	$(INSTALL_PROGRAM) hg_smgrserver$(X) '$(DESTDIR)$(bindir)'/hg_smgrserver$(X)

clean distclean maintainer-clean:
	rm -f hg_smgrserver$(X) $(OBJS) hg_smgrclient_test$(X) hg_smgrclient_test.o
```

For example:

- The command “make” will trigger the “all” target to be executed. This is the default target since we do not specify a target
- “**all**” requires 2 other targets to be done, hg_smgrserver and hg_smgrserver_test, so they will get run as well if they haven’t been created
- hg_smgrserver requires 2 object files and hg_smgrserver_test requires 1 object file. If the dependencies are met the build commands will be run to produce the final executable.

For example:

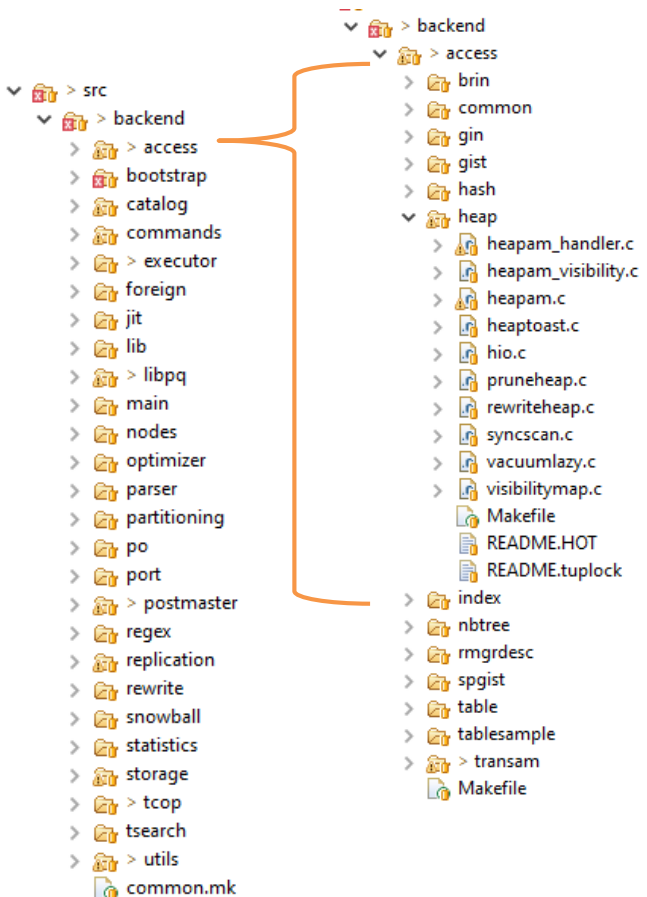
- The command “make install” will trigger the “install” target to be executed, which requires “all” target to be completed already. Then it will run the commands to install
- The command “make clean” will trigger the “clean” target to be executed, which does not depend on anything and it will just run the commands to clean up the code.

Adding A New Backend Module in PostgreSQL

Adding A New Module

How To Add A New Module

- Most likely, the KMS module belongs to the backend of the PostgreSQL system, so it will reside in the “backend” folder.
- But there many sub-folders under the backend folder and each sub-folder contains more folders.
- Where should you place the new module?
- There is no right or wrong answers.
- Put your new module in a place where you feel makes the most sense
- Once you decided on the place, copy the Makefile from another component that is in the same level as yours.
- Change the names to your module.



Connect Your New Module

Bootstrap vs Initialization

- Most of the PG module will have to go through either a bootstrap or initialization process or both.
- Bootstrap process happens during database initialization, (initdb). This means the database server is not really running at this stage.
  - This is only run once during initdb.
  - For example, the XLOG module, (the module responsible for WAL logs), has to go through a bootstrap process to generate the initial WAL files.
- Initialization process happens when a database is started (pg_ctl start). After initialization, the database server remains running.
  - This is run every time the database is started.
  - For example, the XLOG module, is also required to go through an initialization process to register some space from shared memory.

Consideration:

Does your new KMS module requires bootstrap or initialization or BOTH?

Bootstrap Process

Bootstrap

- Bootstrap is controlled by the source file “bootstrap.c”
- Bootstrap process starts from the “**AuxiliaryProcessMain()”** function in src/backend/bootstrap/bootstrap.c
- This may be a good location to add your own bootstrap logic if you need one.
- You simply create a new public function in your module and make sure it gets called during bootstrap process here

Src/backend/bootstrap/bootstrap.c

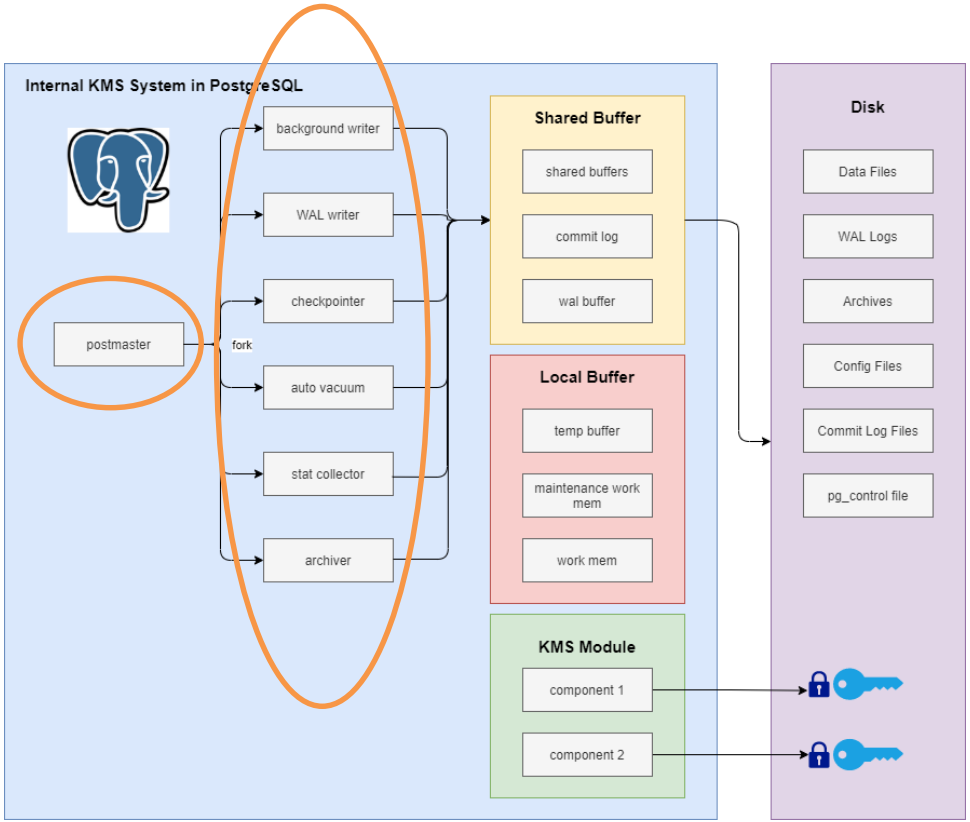
```
197 void
198 AuxiliaryProcessMain(int argc, char *argv[])
199 {
200     char       *progname = argv[0];
201     int         flag;
202     char       *userDoption = NULL;
203
204     /*
205      * Initialize process environment (already done if under postmaster, but
206      * not if standalone).
207      */
208     if (!IsUnderPostmaster)
209         InitStandaloneProcess(argv[0]);
210
```

```
416     /*
417      * XLOG operations
418      */
419     SetProcessingMode(NormalProcessing);
420
421     switch (MyAuxProcType)
422     {
423         case CheckerProcess:
424             /* don't set signals, they're useless here */
425             CheckerModeMain();
426             proc_exit(1);       /* should never return */
427
428         case BootstrapProcess:
429
430             /*
431              * There was a brief instant during which mode was Normal; this is
432              * okay.  We need to be in bootstrap mode during BootStrapXLOG for
433              * the sake of multixact initialization.
434              */
435             SetProcessingMode(BootstrapProcessing);
436             bootstrap_signals();
437             BootStrapXLOG();
438             BootstrapModeMain();
439             proc_exit(1);       /* should never return */
440
441         case StartupProcess:
442             /* don't set signals, startup process has its own agenda */
443             StartupProcessMain();
444             proc_exit(1);       /* should never return */
445
```

Initialization Process

Initialization

- A module's initialization normally happens at 2 places.
- When postmaster starts up
- When a standalone backend starts up



Initialization Process

Postmaster Initialization

- Remember, postmaster is the very first process that gets started.
- It will initialize almost all the backend components during startup.
- It all starts from the "**PostmasterMain()**" function in src/backend/postmaster/postmaster.c
- Here may be a good location to add your own initialization logics if you need one.

tmaster.c

```
574 /*
575  * Postmaster main entry point
576  */
577 void
578 PostmasterMain(int argc, char *argv[])
579 {
580     int         opt;
581     int         status;
582     char       *userDoption = NULL;
583     bool        listen_addr_saved = false;
584     int         i;
585     char       *output_config_variable = NULL;
586
587     InitProcessGlobals();
588
589     PostmasterPid = MyProcPid;
590
591     IsPostmasterEnvironment = true;
592
```

```
1331
1332    /*
1333     * Remove old temporary files.  At this point there can be no other
1334     * Postgres processes running in this directory, so this should be safe.
1335     */
1336    RemovePgTempFiles();
1337
1338    /*
1339     * Initialize stats collection subsystem (this does NOT start the
1340     * collector process!)
1341     */
1342    pgstat_init();
1343
1344    /*
1345     * Initialize the autovacuum subsystem (again, no process start yet)
1346     */
1347    autovac_init();
1348
1349    /*
1350     * Load configuration files for client authentication.
1351     */
1352    if (!load_hba())
1353    {
1354        /*
1355         * It makes no sense to continue if we fail to load the HBA file,
1356         * since there is no way to connect to the database in this case.
1357         */
1358        ereport(FATAL,
1359                (errmsg("could not load pg_hba.conf")));
1360    }
```

Initialization Process

Standalone Backend Initialization

- This is simply a process that is started by the postmaster as a standalone backend and it could be intended to work as one of the backend processes such as “bgwriter”, “checkpointer”, “Stat collector”..etc
- Although they are referred by different names, fundamentally they are all coming from one single binary file called “postgres”
- It all starts from the “**PostgresMain()**” function in src/backend/tcop/postgres.c
- Here may be a good location to add your own initialization logics if you need one. Make sure you do it after the **BaseInit**() as BaseInit() initializes file access, storage manager and buffer pool.

Src/backend/tcop/postgres.c

```
3777 void
3778 PostgresMain(int argc, char *argv[],
3779              const char *dbname,
3780              const char *username)
3781 {
3782     int         firstchar;
3783     StringInfoData input_message;
3784     sigjmp_buf  local_sigjmp_buf;
3785     volatile bool send_ready_for_query = true;
3786     bool        disable_idle_in_transaction_timeout = false;
3787
3788     /* Initialize startup process environment if necessary. */
3789     if (!IsUnderPostmaster)
3790         InitStandaloneProcess(argv[0]);
3791
3792     SetProcessingMode(InitProcessing);
3793
```

```
4000     /* Early initialization */
4001     BaseInit();
4002
4003     /*
4004      * Create a per-backend PGPROC struct in shared memory, except in the
4005      * EXEC_BACKEND case where this was done in SubPostmasterMain. We must do
4006      * this before we can use LWLocks (and in the EXEC_BACKEND case we already
4007      * had to do some stuff with LWLocks).
4008      */
4009 #ifdef EXEC_BACKEND
4010     if (!IsUnderPostmaster)
4011         InitProcess();
4012 #else
4013     InitProcess();
4014 #endif
4015
4016     /* We need to allow SIGINT, etc during the initial transaction */
4017     PG_SETMASK(&UnBlockSig);
4018
```

Shared Memory Initialization

Register Your Own Shared Memory From PG

- We know that PostgreSQL is a multiple-process system, meaning that there are many backend processes running behind the scene responsible for doing different things.
- They rely on shared memory to communicate between themselves to ensure the resource is accessed without conflict.
- Most likely, you will need to have something in the shared memory.

Step 1: Define your struct

This is the structure of data that you would like to share among all backend processes. You can define it within your own module's header file (.h). Note that it is best to use finite data type so we know the size. Avoid using pointers here.

```
typedef struct ESLocalPage
{
    long    max_page_num;
    long    used_page_num;
    long    idle_page;
    LWLock  lock;
} ESLocalPage;
```

Step 2: Declare a **global** variable using your struct

Make sure to declare it in global scale, not within a function.

For Example:

```
static ESLocalPage *es_local_pages = NULL;
```

Shared Memory Initialization

Register Your Own Shared Memory From PG

Step 3: Create a function to return the structure's size.
Remember to also add this function's prototype in your header file so other components can access it.
For example:

```
Size es_page_size(void)
{
    return MAXALIGN(sizeof(ESLocalPage));
}
```

Step 4: Create a function to initialize your structure
This is the function we will use to call PG's shared memory routine (ShmemInitStruct) to register shared memory for our component. You can give it any name in the first parameter of the function and give the size function in step 2 in second parameter.
For example:

```
void es_page_init(void)
{
    bool    found;

    es_local_pages = (ESLocalPage *)
        ShmemInitStruct("ESlocal page", es_page_size(), &found);

    if(!found && es_local_pages)
    {
        memset(es_local_pages, 0, es_page_size());
        es_local_pages->max_page_num = 0L;
        es_local_pages->used_page_num = 0L;
        es_local_pages->idle_page = -1L;
    }
}
```

This is the global variable you declared in step 2

Shared Memory Initialization

Register Your Own Shared Memory From PG

So far, we have 1 structure and 2 function defined in our example. Next, we need to make them called from somewhere.

- es_page_size()
- es_page_init()

Step 5: Go to src/backend/storage/ipc/ipci.c, function CreateSharedMemoryAndSemaphores()

This is the function where PG tries to estimate the total size of shared memory block needed. You can see all other components are requesting share memory here:

Add a new line at the bottom of them after "AsyncShmemSize()", but replacing the second parameter of "add_size" with the size function created in step 3.

For example:

Size = add_size(size, es_page_size());

Src/backend/storage/ipc/ipci.c

```
95  void
96  CreateSharedMemoryAndSemaphores(void)
97  {
98      PGShmemHeader *shim = NULL;

119         size = 100000;
120         size = add_size(size, PGSemaphoreShmemSize(numSemas));
121         size = add_size(size, SpinlockSemaSize());
122         size = add_size(size, hash_estimate_size(SHMEM_INDEX_SIZE,
123                                                  sizeof(ShmemIndexEnt)));
124         size = add_size(size, BufferShmemSize());
125         size = add_size(size, LockShmemSize());
126         size = add_size(size, PredicateLockShmemSize());
127         size = add_size(size, ProcGlobalShmemSize());
128         size = add_size(size, XLOGShmemSize());
129         size = add_size(size, CLOGShmemSize());
130         size = add_size(size, CommitTsShmemSize());
131         size = add_size(size, SUBTRANSShmemSize());
132         size = add_size(size, TwoPhaseShmemSize());
133         size = add_size(size, BackgroundWorkerShmemSize());
134         size = add_size(size, MultiXactShmemSize());
135         size = add_size(size, LWLockShmemSize());
136         size = add_size(size, ProcArrayShmemSize());
137         size = add_size(size, BackendStatusShmemSize());
138         size = add_size(size, SInvalShmemSize());
139         size = add_size(size, PMSignalShmemSize());
140         size = add_size(size, ProcSignalShmemSize());
141         size = add_size(size, CheckpointerShmemSize());
142         size = add_size(size, AutoVacuumShmemSize());
143         size = add_size(size, ReplicationSlotsShmemSize());
144         size = add_size(size, ReplicationOriginShmemSize());
145         size = add_size(size, WalSndShmemSize());
146         size = add_size(size, WalRcvShmemSize());
147         size = add_size(size, ApplyLauncherShmemSize());
148         size = add_size(size, SnapMgrShmemSize());
149         size = add_size(size, BTreeShmemSize());
150         size = add_size(size, SyncScanShmemSize());
151         size = add_size(size, AsyncShmemSize());
```

Shared Memory Initialization

Register Your Own Shared Memory From PG

Step 6: In the same function, add your init function

Scroll down a little further within the same function in step 5, you can see a bunch of init functions for other components

Add a new line at the bottom of them again after AsyncShmemInit(), but with your own init function created in step 4.

For example:

es_page_init();

That Is It!

Src/backend/storage/ipc/ipci.c

```
95  void
96  CreateSharedMemoryAndSemaphores(void)
97  {
98      PGShmemHeader *shim = NULL;

232     InitPredicateLocks();
233
234     /*
235      * Set up process table
236      */
237     if (!IsUnderPostmaster)
238         InitProcGlobal();
239     CreateSharedProcArray();
240     CreateSharedBackendStatus();
241     TwoPhaseShmemInit();
242     BackgroundWorkerShmemInit();
243
244     /*
245      * Set up shared-inval messaging
246      */
247     CreateSharedInvalidationState();
248
249     /*
250      * Set up interprocess signaling mechanisms
251      */
252     PMSignalShmemInit();
253     ProcSignalShmemInit();
254     CheckpointerShmemInit();
255     AutoVacuumShmemInit();
256     ReplicationSlotsShmemInit();
257     ReplicationOriginShmemInit();
258     WalSndShmemInit();
259     WalRcvShmemInit();
260     ApplyLauncherShmemInit();
261
262     /*
263      * Set up other modules that need some shared memory space
264      */
265     SnapMgrInit();
266     BTreeShmemInit();
267     SyncScanShmemInit();
268     AsyncShmemInit();
```

Add A New Config Parameter

Add Your Own Params In Postgresql.conf

- You may also want to define your own configuration parameters for your new component.
- Configuration parameter is also referred as GUC (Global Unified Configuration).
- Follow these steps to add your own parameter

Step 1: Declare Your Global Config Variable

This is the global variable that you have to declare first within your component and initialize it to a default value. You can declare it within your component's source file.

For example:

```
bool        sharedsm_standby = false;
```

Add A New Config Parameter

Add Your Own Params In Postgresql.conf

Step 2: Go to src/backend/utils/misc/guc.c

Depending on the data type of your parameters:

- Bool
- Int
- Real
- String
- Enum

You will need to locate the corresponding structure that defines your parameter.

For example, we want our parameter to be treated as a number so we will go to "ConfigureNamesInt[]" and add our parameter there:

Src/backend/utils/misc/guc.c

```
static struct config_int ConfigureNamesInt[] =
{
    {
        {"archive_timeout", PGC_SIGHUP, WAL_ARCHIVING,
            gettext_noop("Forces a switch to the next WAL file if a "
                         "new file has not been started within N seconds."),
            NULL,
            GUC_UNIT_S
        },
        &XLogArchiveTimeout,
        0, 0, INT_MAX / 2,
        NULL, NULL, NULL
    },


    /* Initialize mode for sharedsm */
    {
        {"sharedsm_standby", PGC_POSTMASTER, REPLICATION_STANDBY,
            gettext_noop("Set the mode for sharedsm, default off."),
            NULL
        },
        &sharedsm_standby,
        0, 0, 1,
        NULL, NULL, NULL
    },

    /* End-of-list marker */
    {
        {NULL, 0, 0, NULL, NULL}, NULL, 0, 0, 0, NULL, NULL, NULL
    }
};
```

Add A New Config Parameter

Add Your Own Params In Postgresql.conf

Step 3: Categorize Your Parameter

Make sure you select the right category for your parameter using these 2 enum values:

- PGC_POSTMASTER means the value can only be set when postmaster starts from config file or the command line. Read about other possible options in guc.h

- REPLICATION_STANDY is the category of the parameter. The example belongs in replication category. Yours may not be. You can put “UNGROUPED” if appropriate. Read about all possible values in guc_tables.h

Src/backend/utils/misc/guc.c

```
    /* Initialize mode for sharedsm */
    {
        {"sharedsm_standby", PGC_POSTMASTER, REPLICATION_STANDBY,
            gettext_noop("Set the mode for sharedsm, default off."),
            NULL
        },
        &sharedsm_standby,
        0, 0, 1,
        NULL, NULL, NULL
    },

    /* End-of-list marker */
    {
        {NULL, 0, 0, NULL, NULL}, NULL, 0, 0, 0, NULL, NULL, NULL
    }
};
```

Add A New Config Parameter

Add Your Own Params In Postgresql.conf

Step 4: Add the new parameter in the default postgresql.conf.sample

- Now that you have added a new GUC in the PostgreSQL system, the next thing you want to do is add your parameter in the default postgresql.conf.sample, so people know about this parameter.
- When you finish initidb, this sample conf file will be used as default in the new cluster
- This file is located in src/backend/utils/postgresql.conf.sample
- Add your parameter to the end of the file
- The hash (#) before the parameter name means the default value (0) will be used. You need to remove this hash, if you want to change the value to something else.

Src/backend/utils/postgresql.conf.sample

```
#------------------------------------------------------------------------------
# CUSTOMIZED OPTIONS
#------------------------------------------------------------------------------

# Add settings for extensions here


#------------------------------------------------------------------------------
# SharedSM OPTIONS
#------------------------------------------------------------------------------
#sharedsm_standby = 0                    # primary: 0, standby: 1
```

That Is It!

The SQL Function

What Is a SQL Function?

- A SQL function is … a function that you can execute on your psql client terminal.
- PostgreSQL has a list of built-in SQL functions that you can execute using the SELECT command.
- In fact, we have used some of these SQL functions in previous lectures… To get the actual data file on disk, remember?
- For your new KMS module, you may also need to define one or more SQL functions to allow user to interact with the module.

For example, to trigger KMS module to do a key rotation.

Examples of SQL Functions:

```
postgres=# select current_date;
 current_date
--------------
 2021-04-16
(1 row)
```

```
postgres=# select pg_relation_filepath('test2');
 pg_relation_filepath
----------------------
 base/12709/16387
(1 row)
```

Add Your Own SQL Function

How To Add One For Your Module?

Step 1: Identify an Unused OID Value

- Remember PostgreSQL references objects using something called a OID value?
- It is basically a number that uniquely identifies an object such as SQL function or tables within PostgreSQL.
- We need to know an OID value that is not currently being used by PG, so we can assign that to our new SQL function.
- To find out, run the script located in src/include/catalog/unused_oids
- And pick a good number to use

For example: 5566

```
caryh@HGPC01:~/highgo/git/projectv/sharedsm$ src/include/catalog/unused_oids
4 - 9
210
270 - 273
357
380 - 381
421
560 - 583
606
702 - 704
760 - 763
784 - 789
811 - 816
1177
1179 - 1180
1382 - 1383
1986 - 1987
2023
2030
2121
2137
2228
3432
3434 - 3435
3998
4035
4142
4187 - 4199
4225 - 4299
4388 - 4399
4532 - 4565
4572 - 4999
5022 - 5027
5032 - 5554
5556 - 5999
6015 - 6099
6103
6105
6107 - 6109
6116
6122 - 9999
```

Add Your Own SQL Function

How To Add One For Your Module?

Step 2: Register your SQL function in pg_proc.dat

- This step is to tells PG catalog that what kind of SQL function you are adding to PG.
- What is the input and output
- The name of the function
- The OID

For example:

a SQL function called pg_rotate_cluster_passphrase that takes no argument and returns a Boolean can be registered as:

```
# function for key managements
{ oid => '5566', descr => 'rotate cluter passphrase',
  proname => 'pg_rotate_cluster_passphrase',
  provolatile => 'v', prorettype => 'bool',
  proargtypes => '', prosrc => 'pg_rotate_cluster_passphrase' },
```

Src/include/catalog/pg_proc.dat

```
#----------------------------------------------------------------------
#
# pg_proc.dat
#    Initial contents of the pg_proc system catalog.
#
# Portions Copyright (c) 1996-2020, PostgreSQL Global Development Group
# Portions Copyright (c) 1994, Regents of the University of California
#
# src/include/catalog/pg_proc.dat
#
#----------------------------------------------------------------------
```

Simply add this block here at the end of the file

Visit the official documentation on pg_proc for more information:
https://www.postgresql.org/docs/current/catalog-pg-proc.html

Add Your Own SQL Function

How To Add One For Your Module?

Step 3: Add your SQL function prototype declaration

- Like a regular C function, a SQL function also needs a prototype declaration so that other modules are aware of this function.
- You can add your prototype in the function manager module in "src/include/utils/fmgrprotos.h"
- A SQL function always has an abstract return type of "datum" and an abstract input argument of "PG_FUNCTION_ARGS"

For example:

```
extern Datum pg_rotate_cluster_passphrase(PG_FUNCTION_ARGS);
```

Src/include/utils/fmgrprotos.h

```
 * fmgrprotos.h

#ifndef FMGRPROTOS_H
#define FMGRPROTOS_H

#include "fmgr.h"

extern Datum heap_tableam_handler(PG_FUNCTION_ARGS);
extern Datum byteaout(PG_FUNCTION_ARGS);
extern Datum charout(PG_FUNCTION_ARGS);
extern Datum namein(PG_FUNCTION_ARGS);
extern Datum nameout(PG_FUNCTION_ARGS);
extern Datum int2in(PG_FUNCTION_ARGS);
extern Datum int2out(PG_FUNCTION_ARGS);
extern Datum int2vectorin(PG_FUNCTION_ARGS);
extern Datum int2vectorout(PG_FUNCTION_ARGS);
extern Datum int4in(PG_FUNCTION_ARGS);
extern Datum int4out(PG_FUNCTION_ARGS);
extern Datum regprocin(PG_FUNCTION_ARGS);
extern Datum regprocout(PG_FUNCTION_ARGS);
extern Datum textin(PG_FUNCTION_ARGS);
extern Datum textout(PG_FUNCTION_ARGS);
extern Datum tidin(PG_FUNCTION_ARGS);
extern Datum tidout(PG_FUNCTION_ARGS);
```

Add your prototype
at the end of this file

Add Your Own SQL Function

How To Add One For Your Module?

Step 4: Implement The SQL Function

- Now, you just need to implement the SQL function and fill it with logics
- This function should be implemented within your module somewhere
- Remember, this example function does not take any input, so we are not doing anything with PG_FUNCTION_ARGS.
- This function returns a Boolean as we have registered to catalog.
- So, in order to return properly, we need to wrap it with a macro, PG_RETURN_BOOL(true). This macro basically makes “datum” to represent a Boolean value as we require.

```
/*
 * SQL function to rotate the cluster passphrase. This function assumes that
 * the cluster_passphrase_command is already reloaded to the new value.
 * All internal keys are wrapped by the new passphrase and saved to the disk.
 * To update all crypto keys atomically we save the newly wrapped keys to the
 * temporary directory, pg_cryptokeys_tmp, and remove the original directory,
 * pg_cryptokeys, and rename it. These operation is performed without the help
 * of WAL.  In the case of failure during rotationpg_cryptokeys directory and
 * pg_cryptokeys_tmp directory can be left in incomplete status.  We recover
 * the incomplete situation by checkIncompleteRotation.
 */
Datum
pg_rotate_cluster_passphrase(PG_FUNCTION_ARGS)
{
    PgAeadCtx   *ctx;
    CryptoKey   newkeys[KMGR_MAX_INTERNAL_KEYS] = {0};
    char        passphrase[KMGR_MAX_PASSPHRASE_LEN];
    uint8       new_kekenc[PG_AEAD_ENC_KEY_LEN];
    uint8       new_kekhmac[PG_AEAD_MAC_KEY_LEN];
    int         passlen;
```

```
    /* Rename to the original directory */
    if (rename(KMGR_TMP_DIR, KMGR_DIR) != 0)
        ereport(ERROR,
                (errmsg("could not rename directory \"%s\" to \"%s\": %m",
                        KMGR_TMP_DIR, KMGR_DIR)));
    fsync_fname(KMGR_DIR, true);

    LWLockRelease(KmgrFileLock);

    pg_free_aead_ctx(ctx);
    PG_RETURN_BOOL(true);
}
```

More On Datum and PG_FUNCTION_ARGS

What are these?

- Datum and PG_FUNCTION_ARGS are just abstractions to all of the data types.
- So instead of worrying about what datatype to use for input or output when designing a PG function, you can just give it either datum or PG_FUNCTION_ARGS
- When you implement the function, you then cast them to what you need.
- Taking “pg_relation_size” SQL function as example:
  - First input argument is cast to OID
  - Second input argument is cast to Text
  - And it returns a 64-bit integer value

```
Datum
pg_relation_size(PG_FUNCTION_ARGS)
{
    Oid         relOid = PG_GETARG_OID(0);
    text       *forkName = PG_GETARG_TEXT_PP(1);
    Relation    rel;
    int64       size;

    rel = try_relation_open(relOid, AccessShareLock);

    /*
     * Before 9.2, we used to throw an error if the relation didn't exist, but
     * that makes queries like "SELECT pg_relation_size(oid) FROM pg_class"
     * less robust, because while we scan pg_class with an MVCC snapshot,
     * someone else might drop the table. It's better to return NULL for
     * already-dropped tables than throw an error and abort the whole query.
     */
    if (rel == NULL)
        PG_RETURN_NULL();

    size = calculate_relation_size(&(rel->rd_node), rel->rd_backend,
                                   forkname_to_number(text_to_cstring(forkName)));

    relation_close(rel, AccessShareLock);

    PG_RETURN_INT64(size);
}
```

Internal KMS Consideration

Define A Scope

What Features To Support?

- We have learned that KMS normally involves in 6 steps in a key's life cycle.
  - Key Generation (required)
  - Key Storage (required)
  - Key Renewal
  - Key Rotation (required)
  - Key Revocation
  - Key Destruction
- I would say 3 out of the above 6 are required as minimum

HIGH GO

融知与行 瀚且高远

THANKS