Tutorial for OS Prerequisites

This set of notes cover the material in system programming in Linux that is useful in Operating Systems. If a student is not well-versed in those materials, this document will help them prepare to be successful in OS.

The document is divided into two sections. Section 1 covers the tools, in particular `Makefile` and `git`. Section 2 covers basic system programming tasks in C, starting with `getopt(3)`, and then, covering process creation and management, exchanging information via shared memory, and performing communication between different processes using signals.

1. Tools

1.1 The `make` Utility

The purpose of the make utility is to ease compilation and linking of different programs. It helps with the maintenance of large programs that may have been broken up into separate compilable files. This helps to reduce the complexity of organizing the code and also reduces the time required for compilation as only those files are recompiled that have been modified.

It is helpful to understand the different phases of compilation and building the executable. In the first phase, the compiler takes the source files (typically indicated by the extension `.c` for C programs) and produces object files with the extension `.o`. This is followed by linking all the object files to produce the executable application. Let us look at the two phases on a Linux machine wuthe the file `hello.c` to produce the executable `hello`. In the first phase, after you have created the file `hello.c` using your favorite editor, you perform the compilation using the command

```
$ gcc -c hello.c
```

The above step creates the object file `hello.o`. In the next step, you create the executable `hello` by using the command

```
$ gcc -o hello hello.o
```

If you are building a library of math functions, you can create each of those functions in separate source files. Let us say that you have the functions square and cube in files `square.c` and `cube.c`, respectively. Also, you are using those functions in the source file `math.c`. Then, the compilation and linking steps, that will result into an executable `math` will be

```
$ gcc -c math.c
$ gcc -c square.o
$ gcc -c cube.c
$ gcc -o math math.o square.o cube.o
```

The compilation can be carried out in any order, that is you may compile `square.c` before or after `cube.c` or `math.c`. But linking is performed only after all the source files have been converted into object files.

The `make` utility automates the above process (and a lot more) by following certain rules that specify dependencies. Like we said earlier, the linking step can be performed only after all the source files have been compiled to create object files. This is another way of saying that the executable `math` is dependent on the existence of the object files `math.o`, `square.o`, and `cube.o`. The rules/dependencies are typically specified in a file named `Makefile`. The `make` utility looks for the file named `Makefile` and follows the rules specified therein to create the executable.

When invoked, the `make` utility examines the modification time of each file and recompiles the files that have been modified since the last build. It does so after sorting the dependency rules to build the executables as specified in `Makefile`. The rules are specified in a non-procedural programming paradigm, such as backtracking used in the logic programming languages.

1.1.1 A Simple Makefile

Let us start looking at the components of a `Makefile` using a very simple version. In this file, we'll build a *target* of the operation, using *prerequisites* or *dependents*; those dependents may have other files as prerequisites.

Each `Makefile` has a default target. A target could be the name of a file generated by a program, for example the name of an executable binary. It could also be the name of an action to carry out, such as clean. A prerequisite is a file used as input to build a target. Thus, a target may have several prerequisites.

The action carried out by `make` is described by a *recipe*. The recipe is specified by one or more commands. Every recipe line, also called a *rule*, is started by a tab character[1]. The recipe creates a target if any of its prerequisites changes; some recipes may not have any prerequisites. Another point to remember is that the same file may be target or prerequisite in different rules. Finally, the comments may be specified by the `#` character, just like in shell scripts. Comments extend to the end of line.

A Makefile consists of two main types of lines, described by the following syntax:

```
target: dep1 dep2 dep2      # Dependency line
      cmd                   # Command line
```

The dependency line shows the dependencies for the target. It is exactly one line long; use of a backslash (`\`) as the last character on the line allows you to extend dependency on multiple lines. The dependencies must be satisfied before a target can be built, possibly by building them as targets in other entries. The list of dependencies may be empty but a target must be followed by the colon character (`:`).

The command lines contain the list of commands to build the target. They must begin with a tab as the first character of the line. You may specify multiple commands on successive lines to be

[1] Starting a recipe line by tab is extremely important. Make sure that your favorite editor actually does not convert tab to spaces. Use the command `cat -A filename` to verify the tabs; this command will display tabs as `^I`.

executed in sequence. You may also specify multiple commands on the same line by separating them with a semicolon (`;`).

Blank lines are allowed in some places. However, they should not separate a dependency line from its commands, neither should they separate the list of commands.

`Makefile` builds the first target by default. You may also specify the default target by the keyword `.DEFAULT_GOAL`.

Any non-existent target, for example `clean`, is treated as a phony target. You may also specify such a target with the keywork `.PHONY`.

In the following, we'll illustrate the `Makefile` by building a utility `mymath` using the files `main.c`, `square.c`, and `cube.c`. We'll assume the existence of a file mymath.h that contains all the prototype functions written by us.

```
.DEFAULT_GOAL
mymath: main.o square.o cube.o
        gcc -o mymath main.o square.o cube.o

main.o: main.c mymath.h
        gcc -c main.c

square.o: square.c mymath.h
        gcc -c square.c

cube.o: cube.c mymath.h
        gcc -c cube.c

.PHONY
clean:
        /bin/rm -f *.o *~ mymath
```

In this file, the first target is `mymath` and it can be built if we have the files `main.o`, `square.o`, and `cube.o`. Each object file is built from the corresponding source file along with the header file. Even though the header file is not used in the command, its use in dependency ensures that the target is recompiled if the header file is modified. Finally, we use the phony target to clean up the directory by issuing the command `make clean`.

1.1.2 Macros or Variables

Macros are used to define variable information, They are entries of the form

```
NAME = text_string
```

Macros are subsequently referred to as `$(NAME)`. The parentheses may be dispensed for macros defined by a single character. The main advantage of using macros is readability and convenience. For example, a macro defined by `DEBUG_FLAG` can allow changing the build from debug to release by changing the value in macro from `-g` to `-O`.

Macros may be defined anywhere in `Makefile` though by convention, they are defined towards the beginning of the file. By convention, macros are defined by using all uppercase letters. A macro definition may be continued on the nxt line by using a backslash as the last character on the line. A macro definition with no string after = is assigned the null string. A macro cannot be redefined within the same file and must be defined before the line in which it is used. The `make` utility expands macros before any other processing.

1.1.3 Suffix Rules

The make utility applies some rules to perform compilation and linking. Suffix rules define the implicit rules. For example, if you want to generate object files from C source files, the rule can be specified as

```
.c.o:
      $(CC) -c $(CFLAGS) $<
```

This rule assumes that all the C source files have an extension `.c`. If the extension is `.cxx`, the first line will change to `.cxx.o`. `#<` is a placeholder for the source file argument; `$@` defines the target for the rule. Our simple `Makefile` using the macros and suffix rules now becomes

```
CC      = gcc         # Default compiler
CFLAGS  = -g          # Compilation flags
TARGET  = math
OBJS    = main.o square.o cube.o quad.o
.SUFFIXES: .c .o

.DEFAULT_GOAL : $(TARGET)
$(TARGET): $(OBJS)
        $(CC) -o $@ $(OBJS)

.c.o:
        $(CC) $(CFLAGS) -c $<

.PHONY: clean
clean:
        /bin/rm -f *.o $(TARGET)
```

1.1.4 Compiling with static archive library

You can use `Makefile` to build and link static archive library. If square and cube functions are to be part of a static archive library, the `Makefile` to compile and build the application is as follows:

```
CC      = gcc
CFLAGS  = -g
TARGET  = math
OBJS    = main.o
LIBOBJS = square.o cube.o
LIBS    = -lmymath
MYLIBS  = libmymath.a
LIBPATH = .
.SUFFIXES: .c .o
```

```
.DEFAULT_GOAL: $(TARGET)
$(TARGET): $(OBJS) $(MYLIBS)
        $(CC) -o $@ -L${LIBPATH} $(OBJS) $(LIBS)

$(MYLIBS): $(LIBOBJS)
        ar -rs $@ $(LIBOBJS)

.c.o:
        $(CC) $(CFLAGS) -c $<

.PHONY: clean
clean:
        /bin/rm -f *.o *~ $(TARGET) $(MYLIBS)
```

1.1.5 Working with dynamically linked libraries

You can also build a dynamically linked library, known as a *shared object* in Linux parlance. These libraries have a suffix of .DLL in Windows but they use the suffix .so in Linux. The advantage is that the application does not need to be recompiled when a shared library is modified, possibly for bug fixes. The new `Makefile` is

```
CC      = gcc
CFLAGS  = -g
TARGET  = math
OBJS    = main.o
LIBOBJS = square.o cube.o foo.o
LIB     = mymath.so

.DEFAULT_GOAL: ALL
ALL:    $(LIB) $(TARGET)

$(TARGET): $(OBJS)
        $(CC) -o $@ $(OBJS) $(LIB)

$(LIB): $(LIBOBJS)
        $(CC) -shared -Wl,-soname,$@ -o $@ $(LIBOBJS)

square.o: square.c
        $(CC) -fpic -c square.c

cube.o: cube.c
        $(CC) -fpic -c cube.c

.c.o:
        $(CC) $(CFLAGS) -c $<

.PHONY: clean
clean:
        /bin/rm -f *.o *~ $(LIB) $(TARGET)
```

An important point to remember when working with shared objects is that you need to modify the environment variable `LD_LIBRARY_PATH`. If your shared object is in the current working directory, you can issue the following command in bash before trying to execute your application:

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:.
```

1.2 Git

Software development involves a lot of trial and error. The utility `git` is a version control system that is designed to help with this. It can be used to save versions of your software at various times, all of which you can revert back to if you like. It is often used in software projects to let multiple developers work together, so it also has the capability to merge different versions together. While `git` can be quite complicated in some cases, its basic usage is pretty straightforward. We will cover these more straightforward features in this section. We will not be discussing the online use of `git` through something like `github`, but instead focus on the local storage of your software versions.

1.2.1 Initializing your project

The first step to using `git` is to create a username and associate an email to that username in your environment. If you intend to use `github` later (or already have a `github` account), it is a good idea to use the same email for both. You do this with the `git` command using the config option

```
git config –global user.name “Your Name”
git config –global user.email “someEmailAddress@address.com”
```

If you go into your home directory now, you will see a new file called `.gitconfig` that will hold these and other `git` settings.

You now want to associate a particular directory with a `git` repository. To do this, go into that directory and then simply type

```
git init
```

This will make a `.git` directory in your project directory that will store your version information and changes.

1.2.2 Staging and committing

To start with, your repository does not track any files. You must tell `git` what files it should keep track of. To do this for a file, you can do with the command

```
git add someFileHere
```

If you want to add all the files in the current directory, you can do

```
git add .
```

At any time you can type

```
git status
```

To see what files are being tracked and whether they have been changed. Note though, that at this point we have not saved any of the changes to our repository. To do this, again use the `git` command to commit your changes. Make sure you associate a useful message with the commit to make it easier to revert to a particular version later.

```
git commit -m "Some useful message here"
```

Each commit is associated with a unique hash. You can use those hashes to refer to particular versions. For example, suppose a previous commit had a hash of 263dced. You could compare it to the current version you are working with using:

```
git diff 263dced
```

The output of this command will tell us what has changed. Portions of code that do not start with a + or a – have remained unchanged. Any lines starting with a + or a – are lines that are coming from one of the two different versions. It can seem a bit confusing at first to compare versions, but after a few uses it will be straightforward.

At any time, you can get a list of all your different commits and all their associated hashes using:

```
git log
```

1.2.3 Checking out or resetting

Each time we perform a commit, we have essentially created a waypoint that we can go back to. If we want to go back to a particular version, we reference its hash. You do not need to use the whole hash, just enough so it is unique. Suppose I want to go back to my code that I committed with a hash of 263dced:

```
git checkout 263dced
```

Using hashes can be confusing. We can tag a particular version ourselves (essentially naming it) using the tag option to git. Suppose I wanted to name my particular version version23:

```
git tag version23
```

If you have made a change to a particular file and you realize it was a mistake, you can reset that file to how it started in your current version:

```
git reset someFile
```

While there is much more to `git` than this, for example you can checkout temporary branches and then even merge separate branches together, this should be enough for you to start using `git`.

2. C Programming

2.1 Command line parsing (`getopt`)

The `getopt` function is used to parse command line options provided to your application. Its prototype is given by

```
int getopt ( int argc, char * const argv[], const char * optstring );
```

The parameters `argc` and `argv` are the same as provided to the `main` function as your application is started. Within `argv`, the tokens indicating the optional items are prefixed by the character `'-'` or the string `"--"`. Each character following `'-'`, up to the space character, represents an option. Repeated calls to `getopt` return each of the option characters, whether part of the same string or specified separately. This means that the command "`ls -li`" is the same as "`ls -i -l`". An added advantage is that the options may be specified in any order, or not at all.

The function `getopt` uses an implicit global variable `optind` that is initialized to 1 at the beginning of the application. `optind` points to the index within the array of string tokens `argv`. Notice that `argv[0]` contains the name of the executable itself and hence, `optind` points to the first parameter on the command line. The variable `optind` is modifiable in the application and may be reset to 1 to start a rescan of parameters.

If `getopt` finds an option character, it returns that character and updates `optind` as well as a static variable `nextchar`. The subsequent call to `getopt` resumes with the following option character or `argv` token. If there is no more option character, `getopt` returns -1 and `optind` points to the first token in `argv` that is not an option.

The parameter `optstring` is used to specify the set of valid option characters in a string. Any option that needs to be followed by an argument (for example, `-n 10` such that the option `n` is assigned a value `10`), is followed in `optstring` by the character `':'`. This option argument is captured in a string variable `optarg`.

If `getopt` encounters an option character that is not present in `optstring`, it stores the character in a variable `optopt` and returns the character `'?'`. This allows for the application to gracefully degrade; the application can choose to abort after printing an error message, or ignore the error condition by setting the variable `opterr` to 0.

When `getopt` finds a valid option character, it returns that character. If all the options have been parsed, `getopt` returns `-1`.

2.1.1 Example code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <linux/limits.h>
```

```
// This driver program is used to describe collecting a set of options.  It
is
// picked from a code that I wrote to display image files contained in the
// specified directory.  If no directory is specified, it will display the
// images in the current directory.  The number of rows and columns specify
// the maximum size for each image.
//
// It is a good programming practive to collect the options in a structure
// defined for the purpose.  It keeps the code clean.  If you do not use a
// structure, the options will be stored in variables with possibly global
// scope.

// Structure to define the options for the applications.

typedef struct
{
    int       rows;           // Number of rows; default 480
    int       cols;           // Number of columns; default 640
    char *    dir;            // Directory containing pictures; default . or
                              // current directory
} options_t;


void print_usage ( const char * app )
{
    fprintf (stderr, "usage: %s [-h] [-r nrows] [-c ncols] [-d dir]\n",\
                app );
    fprintf (stderr, "    nrows is the number of columns\n");
    fprintf (stderr, "    ncols is the number of rows\n");
    fprintf (stderr, "    dir is the directory containing images\n\n");
    fprintf (stderr, "Default image dimensions are 640x480\n");
    fprintf (stderr, "Default directory is the current directory\n");
}

int main(int argc, char *argv[])
{
    options_t options;

    // Set default values

    options.rows = 480;
    options.cols = 640;
    options.dir = ( char * ) malloc ( MAX_CANON * sizeof ( char ) );
    strcpy ( options.dir, "." );

    const char optstr[] = "hr:c:d:";

    char opt;
    while ( ( opt = getopt ( argc, argv, optstr ) ) != -1 )
    {
        switch ( opt )
        {
        case 'h':
            print_usage ( argv[0] );
            return ( EXIT_SUCCESS );

        case 'c':
```

```
            options.cols = atoi ( optarg );
            break;

        case 'r':
            options.rows = atoi ( optarg );
            break;

        case 'd':
            strcpy ( options.dir, optarg );
            break;

        default: /* '?' */
            printf ( "Invalid option %c\n", optopt );
            print_usage ( argv[0] );
            return ( EXIT_FAILURE );
        }
    }


    // Display the collected (or default) options

    printf ( "\tValue of options: \n" );
    printf ( "\tImage size: %dx%d\n", options.cols, options.rows );
    printf ( "\tDirectory: %s\n", options.dir );

    // Display the remaining parameters

    int i;
    printf ( "List of parameters:\n" );
    for ( i = optind; i < argc; i++ )
    {
         printf ( "\tParameter %d: %s\n", i - optind + 1, argv[i] );
    }

    /* Other code omitted */

    exit(EXIT_SUCCESS);
}
```

2.2 Processes (fork/exec/wait)

In this section we will cover the three main calls that are used to create new processes and run new executable files from another C/C++ program.

2.2.1 Creating a new process with fork

The `fork()` call lets you spin off another process to execute in the background that is an almost exact copy of the original. After this call, you will now have two processes running that both continue after the call. One, the parent, gets returned the process id of the child from the `fork()` call. The other process, the child process, gets a return value of 0 from the call. It is possible for fork to fail, in which case it returns a -1.

After the `fork()` call, as they are copies of one another (except for the single value of the return of `fork()`), they share all the initial values of variables, arrays, etc. However, they have *separate*

memory spaces after the `fork()`, so you cannot communicate using normal variables. If you want to communicate, you must use additional methods like shared memory (discussed later).

The following code will fork off a child and then have both parent and child output to screen.

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char** argv) {
   pid_t childPid = fork(); // Child process splits from
   if (childPid == 0) {
      printf("I am a child! My parent's PID is %d, and my PID is %d\n",
             getppid(), getpid());
   } else if (childPid > 0) {
      printf("I'm a parent! My pid is %d, and my child's pid is %d \n",
             getpid(), childPid);
   }
   else {
      perror("Fork failed\n");
   }
   return EXIT_SUCCESS;
}
```

The `getpid()` call is used by a process to get its own process id. The `getppid()` call returns the process id of a processes parent.

This ability to spin off multiple processes can be used to split up a task over multiple processes.

2.2.2 Simple synchronization with wait()

After the execution of a `fork()` call, processes are not synchronized in any way. One way to do very basic synchronization is with the `wait()` call. The wait() call sleeps the calling process until one of the processes children terminate and returns pid of the process that had terminated. In particular, when that child process terminates and sends a SIGCHILD signal to the parent. The simplest use case of this is when you want a parent process to suspend itself until a child process that it generated with `fork()` has terminated. It is also possible to get the return status of the child that has terminated, as well as its pid in the case where a parent might want to know which of its children have terminated.

By default `wait()` is blocking, that is it simply suspends the process until it receives the SIGCHILD signal. However, we will see later on that it is possible to do a nonblocking version that simply polls the system to see if a child has terminated and if so, returns which one did so.

In the following code below, a process forks off a child and then lets its child execute and finish first, before getting its status. Note that we use the macro WEXITSTATUS to convert the returned status code.

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char** argv) {
   pid_t childPid = fork(); // Child process splits from
```

```
   if (childPid == 0) {
      printf("I am a child! My parent's PID is %d, and my PID is %d\n",
             getppid(), getpid());
   } else if (childPid > 0) {
      printf("Let us wait for child to terminate\n");
      int status;
      int terminatedPid = wait(&status);

      printf("I'm a parent! Child with pid of %d has terminated!\n",
             getpid(), terminatedPid);
      printf("It had a return status of %d\n",WEXITSTATUS(status));
   }
   else {
      perror("Fork failed\n");
   }
   return EXIT_SUCCESS;
}
```

It is also possible to do a non-blocking wait() call using the more fully featured waitpid() call. This is often useful when you want to do some additional work, while periodically checking to see if one of the child processes has terminated.

The waitpid() call takes in 3 arguments. By default the waitpid call waits for any terminated child, but this can be modified by the first argument to wait for a particular set of processes. The second argument is for the status code return. The last argument specifies whether it is blocking or not, using constants such as WNOHANG.

For example, the line:

```
int pid = waitpid(-1, &status, WNOHANG);
```

would specify looking for any child process to terminate (-1 in first argument) and doing so in a nonblocking fashion (WNOHANG as last argument) and would return the status code of the terminating child in `status`. In the case that a child had terminated, the variable pid would get the process id of the child that had terminated. In the case where it had not terminated, the `pid` would equal 0.

The following code launches a child process and then the parent continues to poll for the child process to finish while periodically doing some outputs itself:

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char** argv) {
   pid_t childPid = fork(); // Child process splits from
   if (childPid == 0) {
      int i;
      for (i = 0; i < 10; i++) {
         printf("Iteration %d in child, my parent PID is %d, my PID is %d\n",
                i,getppid(), getpid());
         sleep(1);
      }
   } else if (childPid > 0) {
```

```
      printf("Let us periodically see if child has terminated\n");
      int status;

      int i;
      int childFinished = 0;
      do {
         childFinished = waitpid(-1,&status,WNOHANG);
         if (!childFinished) {
            printf("Im the parent, child still has not terminated!\n");
            sleep(1);
         }

      } while (!childFinished);
      printf("I'm a parent! Child with pid of %d has terminated!\n",
            getpid(), childFinished);
      printf("It had a return status of %d\n",WEXITSTATUS(status));
   }
   else {
      perror("Fork failed\n");
   }
   return EXIT_SUCESS;
```

2.2.3 Launching separate executables

While the `fork()` call creates a copy of a child process, the `exec()` family of functions replaces a process with a completely separate process. Used together, `fork()` and `exec()` can be used to launch completely separate executables. In fact, that is exactly how the shell launches separate executables when you do a command.

The `exec()` family replaces the process that used it with a completely separate process. *Note that this means that the process that runs an exec() call that is successful is no more!* If successful, the `exec()` call will not return any value as the code that calls it is gone. Due to this, a call to `fork()` is usually used to create a child and parent and then the child process then does an `exec()` call.

As the `exec()` call is attempting to run another executable, it is necessary to pass into the call the location of that executable, the name of the executable and the runtime arguments to that executable. While the `exec()` family consists of 6 different functions (execl, execlp, execle, execv, execvp and execvpe), they only differ in how you send the above information into that function. You should select the particular exec function that is easiest for you to do that based on the format of the information.

The following code does a `fork()` to generate a child process. The child process then does an `exec()` to execute a process called "./child" and gives it a set of arguments. The function `execlp()` is used for the call, but also commented out is another possible call using the `execvp()` function:

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
```

```
#include<stdlib.h>
int main(int argc, char** argv) {
   pid_t childPid = fork(); // This is where the child process splits from
the parent
   if (childPid == 0) {
      printf("Child but copy of parent! Parent PID is %d, and my PID is
%d\n",
            getppid(), getpid());
      char* args[] = {"./child", "Hello",
         "there", "exec", "is", "neat", 0};

      //execvp(args[0], args);

execlp(args[0],args[0],args[1],args[2],args[3],args[4],args[5],args[6]);

      fprintf(stderr,"Exec failed, terminating\n");
      exit(1);

   } else {
      printf("I'm a parent! My pid is %d, and my child's pid is %d \n",
            getpid(), childPid);
      //sleep(1);
      wait(0);
   }
   printf("Parent is now ending.\n");
   return EXIT_SUCCESS;
}
```

Note that immediately following the `exec()` call is a simple printf of an error message, as if code ever gets past the exec call, it means the `exec()` call failed. It is very common for coders to make mistakes setting up an `exec()` call, so ensure you have this error checking.

The following code is what made up the ./child executable that is executed from the child generated by the above code.

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char** argv) {
   printf("Hello from Child.c, a new executable!\n");
   printf("My process id is: %d\n",getpid());
   printf(" I got %d arguments: \n", argc);

   int i;
   for (i =0; i < argc; i++)
      printf("|%s| ", argv[i]);
   printf("\nChild is now ending.\n");

   sleep(3);
   return EXIT_SUCCESS;
}
```

One key point to remember with the `exec()` family is that you are REPLACING the previous running process with the new process. This means the pid of the new executable is the same as the process that did the `exec()` call. Also, always follow up an `exec()` call with an error check and an exit call. After all, if you wanted to do an `exec()` call and it failed, you were expecting it to replace that process so you definitely do not want it to keep running!

2.3 Shared memory

Normal dynamic memory (allocated using `malloc()` or `new()`) is only accessible by one process. Shared memory can be accessed by multiple processes and can be used for communication between separate processes. In this section I will be describing how to allocate and use shared memory. Keep in mind that this is far from a comprehensive guide on shared memory. This is more a guide on how to quickly set up a section of shared memory and use it between processes.

2.3.1 Allocating shared memory

The first thing we need to do is pick a shared memory key. This key will uniquely identify a particular segment of shared memory and must be known by any process. This can be done in two ways, either by picking a number ourselves or by creating that number based on a file.

The easiest way to create a shared memory key is simply to pick a number. Remember though, we want to make sure that some other user doesn't pick the same number, so we want to pick something fairly large. For example, if we wanted our shared memory to have a key of 2031974, we would first save that value into a variable, like:

```
const int sh_key = 2031974;
```

If we wanted to instead let the system uniquely identify a number with a particular file location, we can use `ftok()`. This library call always generates the same number given the same file on a system. Let us say I have a file main.c in my directory, I can use the following command to generate our shared memory key.

```
const int sh_key = ftok("main.c",0);
```

Pretty straightforward. The 0 following the "main.c" is used to generate other unique keys after the first if I want to create several segments.

Now we have our key, let us allocate some memory associated with that key! To do that, we use `shmget()`, which will return a shared memory id associated with the memory allocated with our key. Using `shmget()` we can ask for a specific number of bytes and also associate user/group/other permissions with this segment. For example, to allocate enough space for 10 integers, we could do the following:

```
int shm_id = shmget( sh_key , sizeof(int) * 10 , IPC_CREAT | 0666  );
if (shm_id <= 0) {
   fprintf(stderr,"Shared memory get failed\n");
   exit(1);
}
```

Here we see that we request memory associated with our `sh_key`, of a particular quantity. Then we specify some flags. In this case, we are saying "create it anew" with `IPC_CREAT` and then `0666` are the permissions on our memory. In this case, 666 means that user/group/other have read and write permissions. After a call to `shmget()`, it is always a good idea to do error checking also, which we do right after.

2.3.2 Attaching to shared memory

One more step! Now we need to associate a pointer to this `shm_id` so we can use it. For that we use `shmat()`, as in this stage we are going to attach this memory we allocated to our process and point a pointer at the memory.

```
int *shm_ptr = shmat( shm_id , 0 , 0 );
if (shm_ptr <= 0) {
   fprintf(stderr,"Shared memory attach failed\n");
   exit(1);
}
```

At this point we are set! We have a pointer to our shared memory segment and can use it just like any other dynamically allocated memory. We can use it as an array or in any other way that we want.

2.3.3 Detaching and freeing up shared memory

Note though that this memory *will not be deallocated* when the process terminates. Instead, you must terminate it yourself or it will stick around in the system, thereby wasting system resources. At the end of your program, you should make sure to deallocate it. The first step is to "undo" our attachment that we did with `shmat()`. Then we want to free up the shared memory.

```
shmdt( shm_ptr );    // Detach from the shared memory segment
shmctl( shm_id, IPC_RMID, NULL ); // Free shared memory segment shm_id
```

If you forget (or your program crashes or terminates) before freeing up the shared memory, you can also manually free it up from the shell. You can get a list of shared memory using the command:

```
ipcs
```

If you see your username, then you can manually free any segments by using the command:

```
ipcrm -m shmid
```

Using the shmid you saw using ipcs.

2.3.4 Putting it all together

Let us now look at a small example that shows a process executing a fork() to create a child process and then communicating with it using shared memory.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void parent();
void child ();

#define SHMKEY  859047     /* Parent and child agree on common key.*/
#define BUFF_SZ sizeof ( int )

int main()
{
    switch ( fork() )
    {
    case -1:
        fprintf(stderr,"Failed to fork\n");
        return ( 1 );
    case 0:
        child();
        break;
    default:
        parent();
        break;
    }
    return ( 0 );
}

void parent()
{
    // Get shared memory segment identifier
    int i;

    int shmid = shmget ( SHMKEY, BUFF_SZ, 0777 | IPC_CREAT );
    if ( shmid == -1 ) {
       fprintf(stderr,"Parent: ... Error in shmget ...\n");
       exit (1);
    }

    // Get the pointer to shared block
    char * paddr = ( char * )( shmat ( shmid, 0, 0 ) );
    int * pint = ( int * )( paddr );

    for ( i = 0; i < 10; i++ ) {
       sleep ( 2 );
```

```
        *pint = 10 * i ;              /* Write into the shared area. */
        printf("Parent: Written Val.: = %d\n",*pint);
    }
    shmdt(pint);
    shmctl(shmid,IPC_RMID,NULL);
}

void child()
{
    int i;
    sleep ( 5 );
    int shmid = shmget ( SHMKEY, BUFF_SZ, 0777 );

    if ( shmid == -1 ) {
        fprintf(stderr,"Child: ... Error in shmget ...\n");
        exit ( 1 );
    }

    int * cint = ( int * )( shmat ( shmid, 0, 0 ) );

    for ( i = 0; i < 10; i++ ) {
        sleep ( 1 );
        printf("Child: Read Val. = %d\n",*cint);
    }
    shmdt(cint);
}
```

Note that the child process should not free up the memory, but instead simply detaches from it. The parent should free the memory after all the children have detached and terminated.

2.4 Signal handling

Signal handling in Unix/Linux provides a mechanism for asynchronous communication between independent processes. Since signals are asynchronous, the process should set up the mechanism to respond to signals (signal handler function), preferably as the process starts. The signal handler will specify the action to be taken when a signal is received. There may be a default action associated with a signal such as terminating the process in response to the signal `CTRL-C`, or suspending it in response to `CTRL-Z`, sent by the shell. The process may decide to ignore the signal, or perform the default action, or perform a specified action.