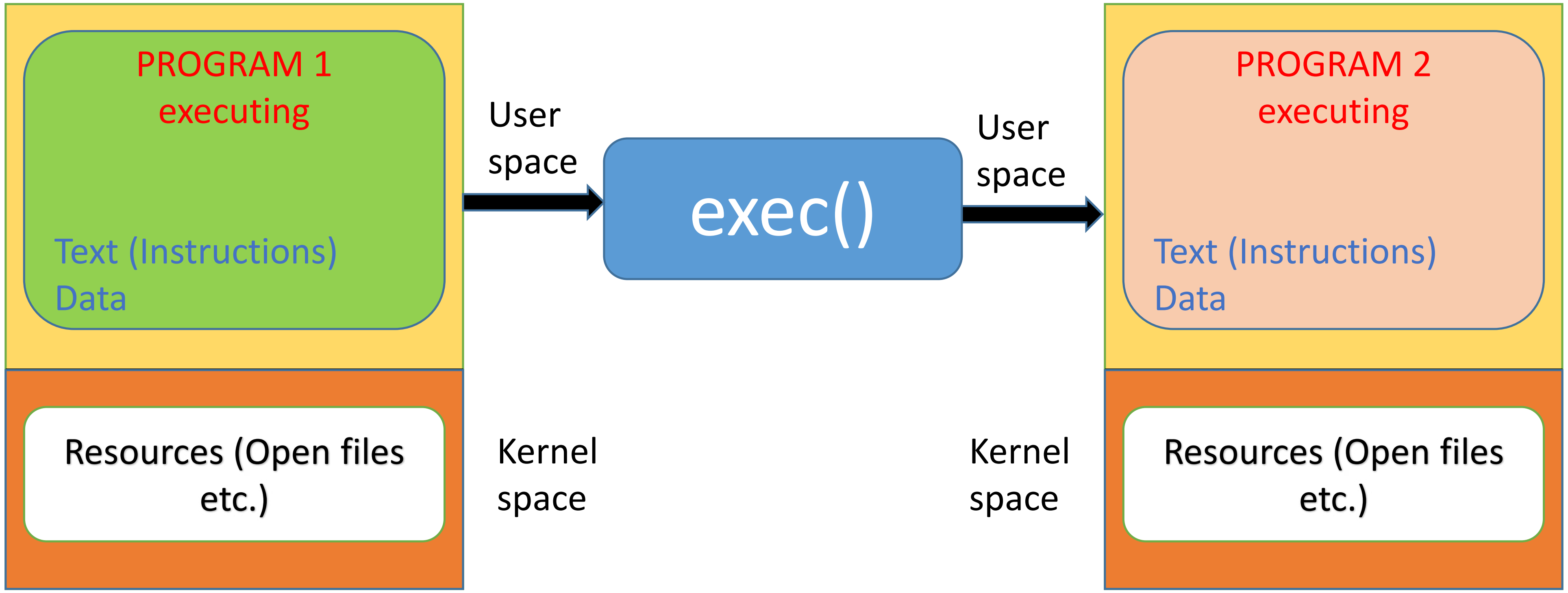
Introducing exec() system calls

- The exec() functions (there are more than one) are a family of functions that execute some program within the current process space.
- So if you write a program that calls one of the exec functions, as soon as the function call succeeds the original process gets replaced with whatever program you have asked exec() to execute.
    - Usually used in conjunction with a fork() call, but not mandatory.
- You would typically fork a child process, and then call exec() from within the child process, to execute some other program in the new process entry created by fork().
- The exec() system call used after a fork() system call helps one of the two processes to replace the memory space with a new program.
- The exec() system call loads a binary file into memory (destroying image of the program containing the exec() system call).
- Within the exec() family there are functions that vary slightly in their capabilities.

Working of exec() system call



exec() family of System Calls

When fork() creates a child process with a copy of same code, data etc as parent process but if you need to run another process as child process then →

A process may replace its current code, data, and stack with those of another executable by using one of the “exec()” family of system calls

When a process executes an “exec()” system call, its PID and PPID numbers stay the same - only the code that the process is executing changes.

System Call:

int execl(const char* path, const char* arg0, const char* arg1,…, const char* argn, NULL)

int execv(const char* path, const char* argv[])

int execlp(const char* path, const char* arg0, const char* arg1, …, const char* argn, NULL)

int execvp(const char* path, const char* argv[])

The “exec()” family of system calls replaces the calling process’ code, data, and stack with those of the executable whose pathname is stored in path.

execl() VS. execlp() calls

- execl(): It permits us to pass a list of command line arguments to the program to be executed.
  - The list of arguments is terminated by NULL.
  - Usage: `execl("/bin/ls", "ls", "-l", NULL);`
- execlp(): It does same job as execl() except that it will use environment variable PATH to determine which executable to process.
  - Thus a fully qualified path name would not have to be used.
  - execlp() can also take the fully qualified name as it also resolves explicitly.
  - Usage: `execlp(“ls”, “ls”, “-l”, NULL);`
  - Note we have not used “/bin/ls” as in previous execl() call

execv() VS. execvp() calls

- execv():
  - It does same job as execl() except that command line arguments can be passed to it in the form of an array of pointers to string.
  - Usage:

```
char *argv[] = ("ls", "-l", NULL);
execv("/bin/ls", argv);
```

- execvp():
  - It does same job as execv() except that it will use environment variable PATH to determine which executable to process.

Difference in exec() System Calls

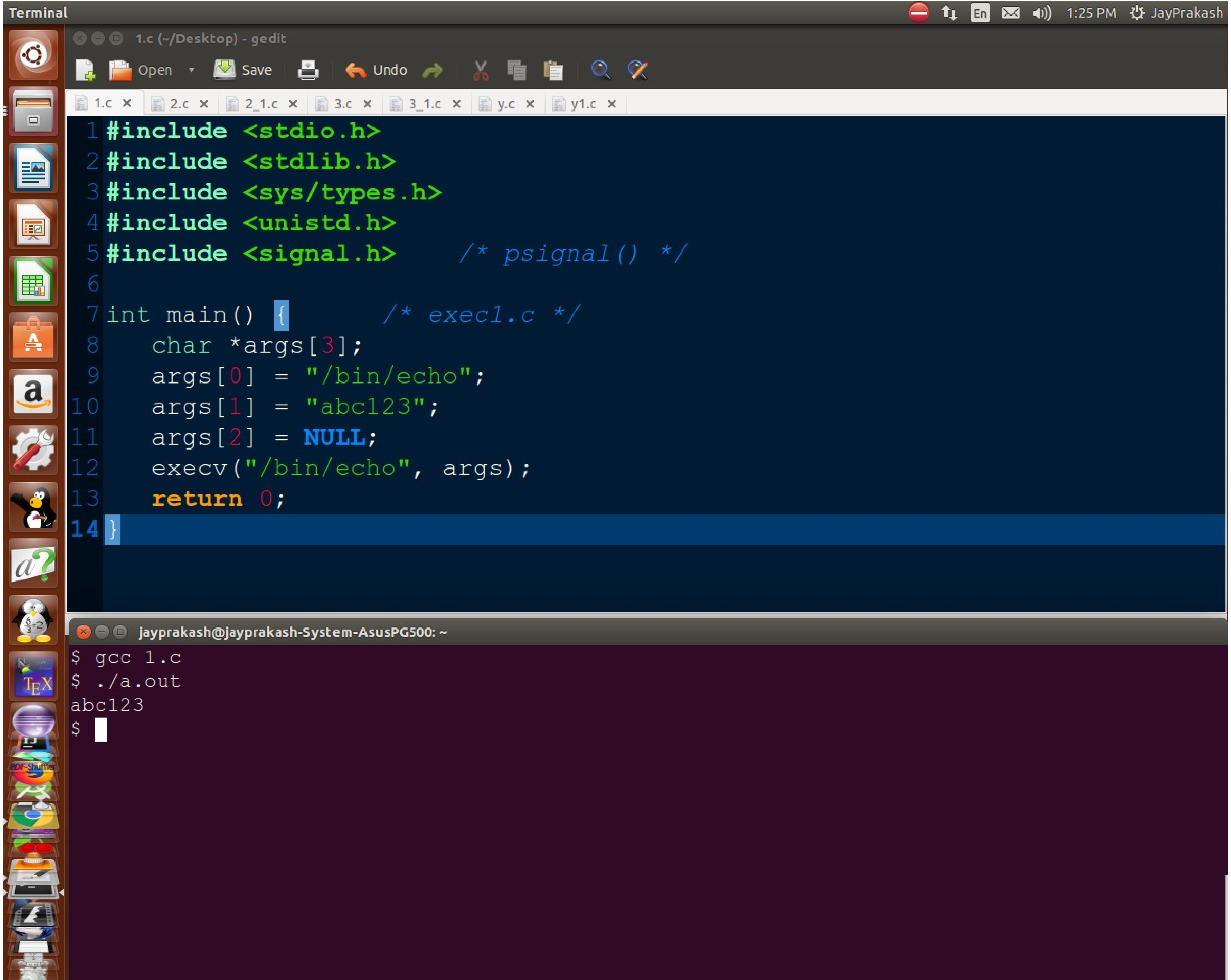
- “execlp()” and “execvp()” use the $ PATH environment variable to find path.
  - If the executable is not found, the system call returns a value of -1; otherwise, the calling process replaces its code, data, and stack with those of the executable and starts to execute the new code.
- “execl()” and “execlp()” invoke the executable with the string arguments pointed to by arg1 through argn.
  - arg0 must be the name of the executable file itself, and the list of arguments must be null terminated.
- “execv()” and “execvp()” invoke the executable with the string arguments pointed to by argv[1] to argv[n], where argv[n+1] is NULL.
  - argv[0] must be the name of the executable file itself.

System call exec() Example

the program displays a small message and then replaces its code with that of the “ls”.

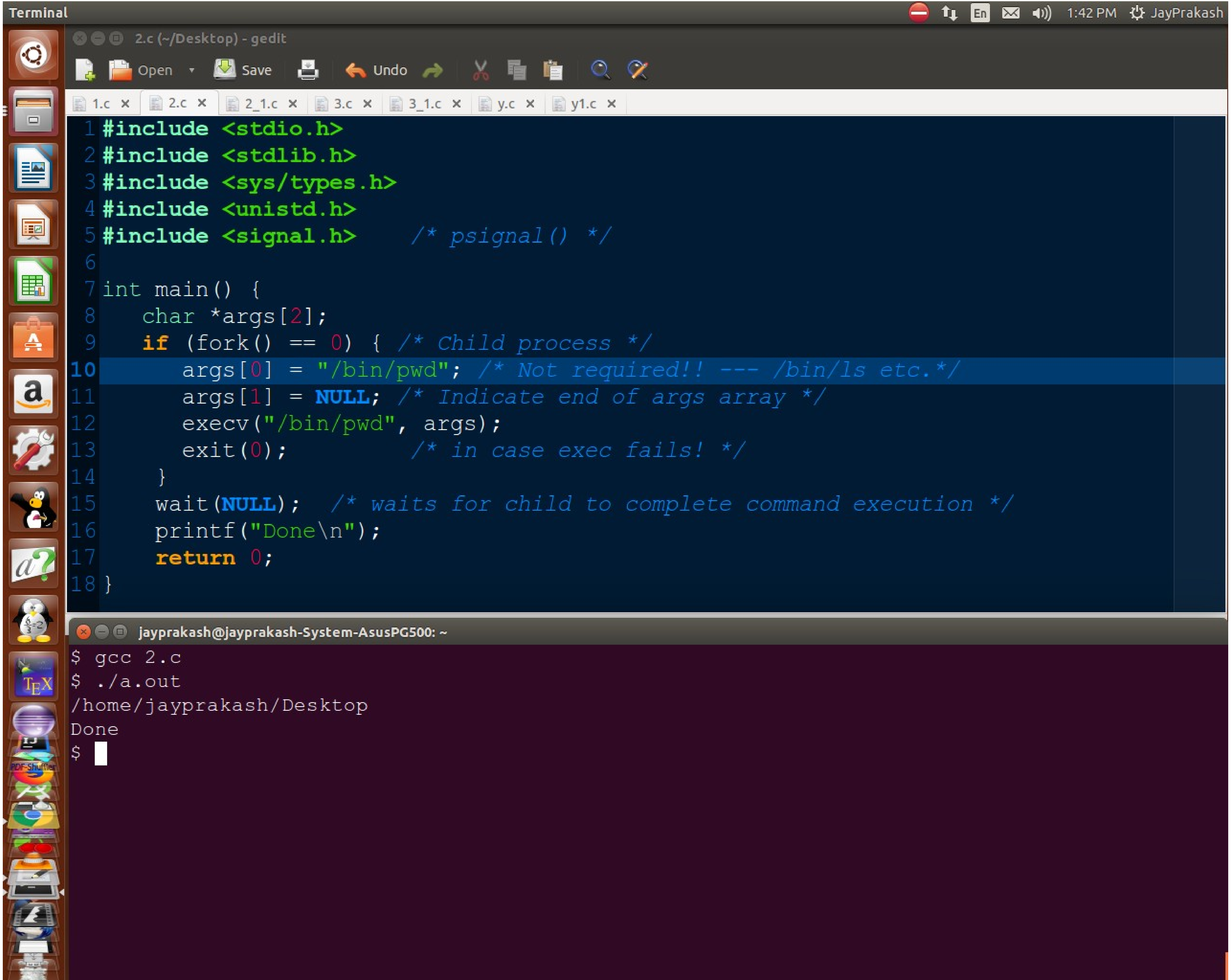
```
#include <stdio.h>
main()
{
   printf(“I’m process %d and I’m about to exec an ls -l \n”, getpid() );
   execl( “/bin/ls”, “ls”, “-l”, NULL ); /* Execute ls */
   printf(“This line should never be executed \n”);
}
```

```
$ myexec ---> run the program.
I’m process 13623 and I’m about to exec an ls -l
total 125
-rw-r--r-- 1 glass   277 Feb 15 00:47 myexec.c
-rwxr-xr-x 1 glass 24576 Feb 15 00:48 myexec
```



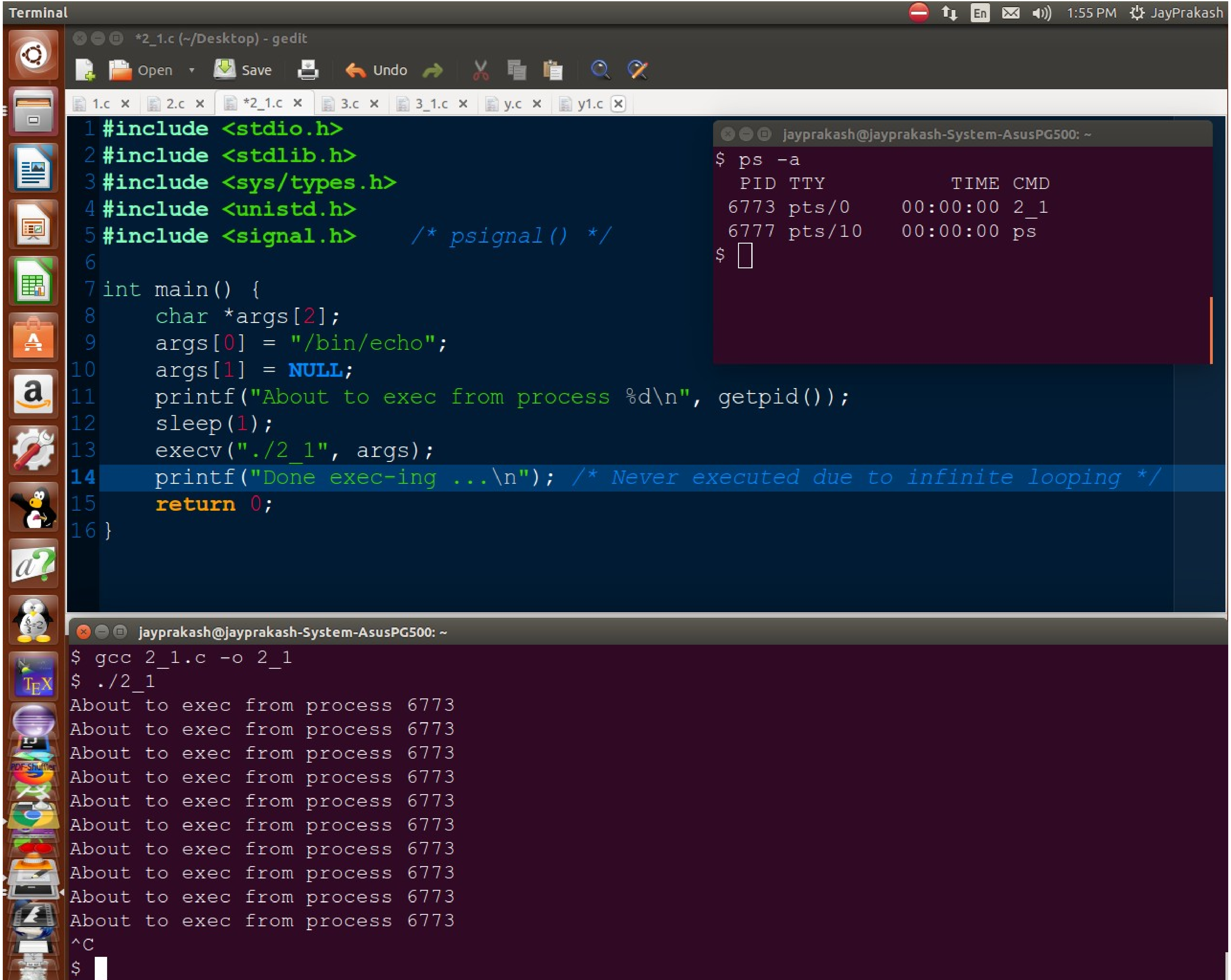
```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>    /* psignal() */

int main() {      /* execl.c */
   char *args[3];
   args[0] = "/bin/echo";
   args[1] = "abc123";
   args[2] = NULL;
   execv("/bin/echo", args);
   return 0;
}
```

```
$ gcc 1.c
$ ./a.out
abc123
$ 
```



```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>     /* psignal() */

int main() {
   char *args[2];
   if (fork() == 0) { /* Child process */
      args[0] = "/bin/pwd"; /* Not required!! --- /bin/ls etc.*/
      args[1] = NULL; /* Indicate end of args array */
      execv("/bin/pwd", args);
      exit(0);          /* in case exec fails! */
    }
    wait(NULL);  /* waits for child to complete command execution */
    printf("Done\n");
    return 0;
}
```

```
$ gcc 2.c
$ ./a.out
/home/jayprakash/Desktop
Done
$ 
```



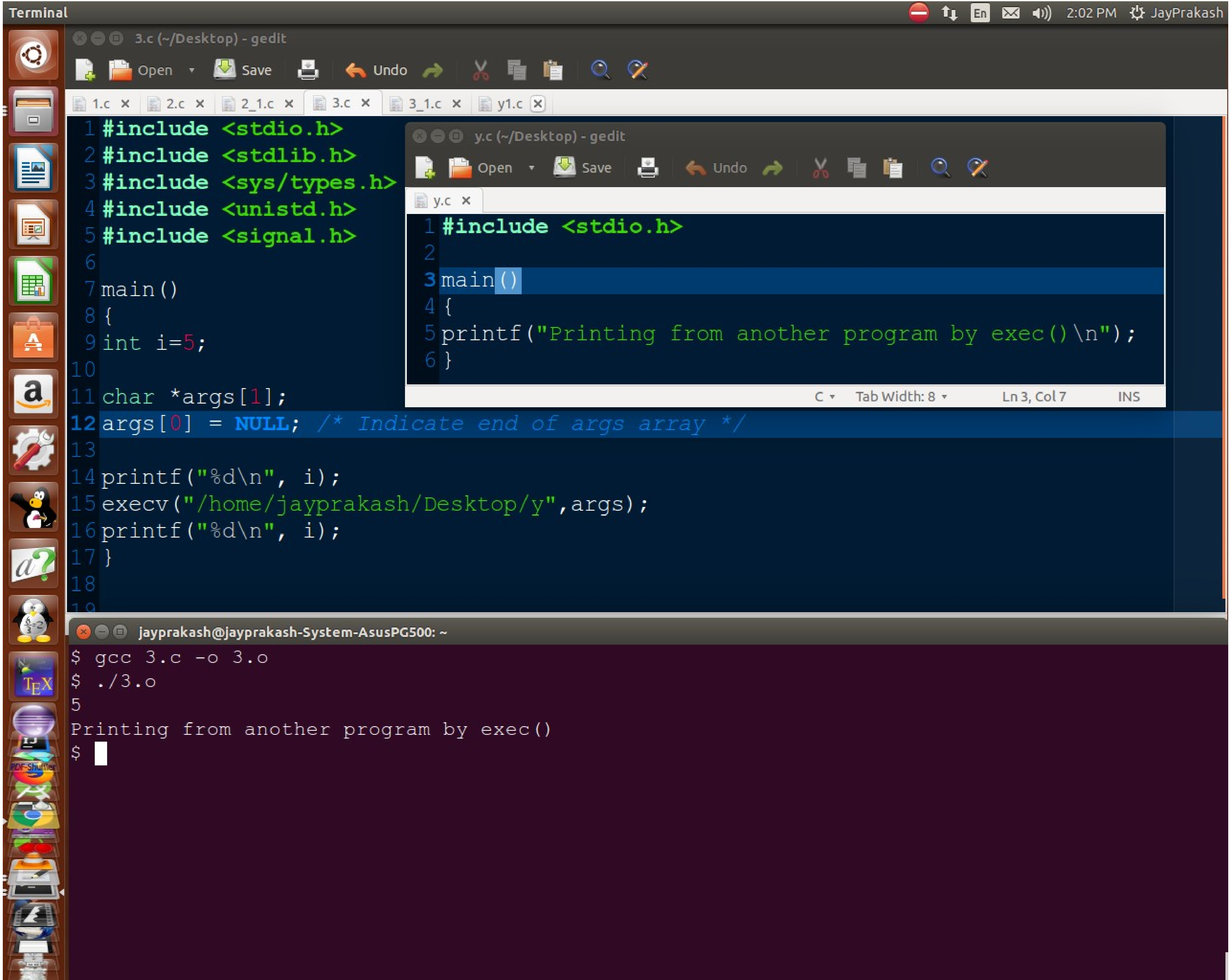
```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>    /* psignal() */

int main() {
    char *args[2];
    args[0] = "/bin/echo";
    args[1] = NULL;
    printf("About to exec from process %d\n", getpid());
    sleep(1);
    execv("./2_1", args);
    printf("Done exec-ing ...\n"); /* Never executed due to infinite looping */
    return 0;
}
```

jayprakash@jayprakash-System-AsusPG500: ~

```
$ ps -a
  PID TTY          TIME CMD
 6773 pts/0    00:00:00 2_1
 6777 pts/10   00:00:00 ps
$ 
```

jayprakash@jayprakash-System-AsusPG500: ~

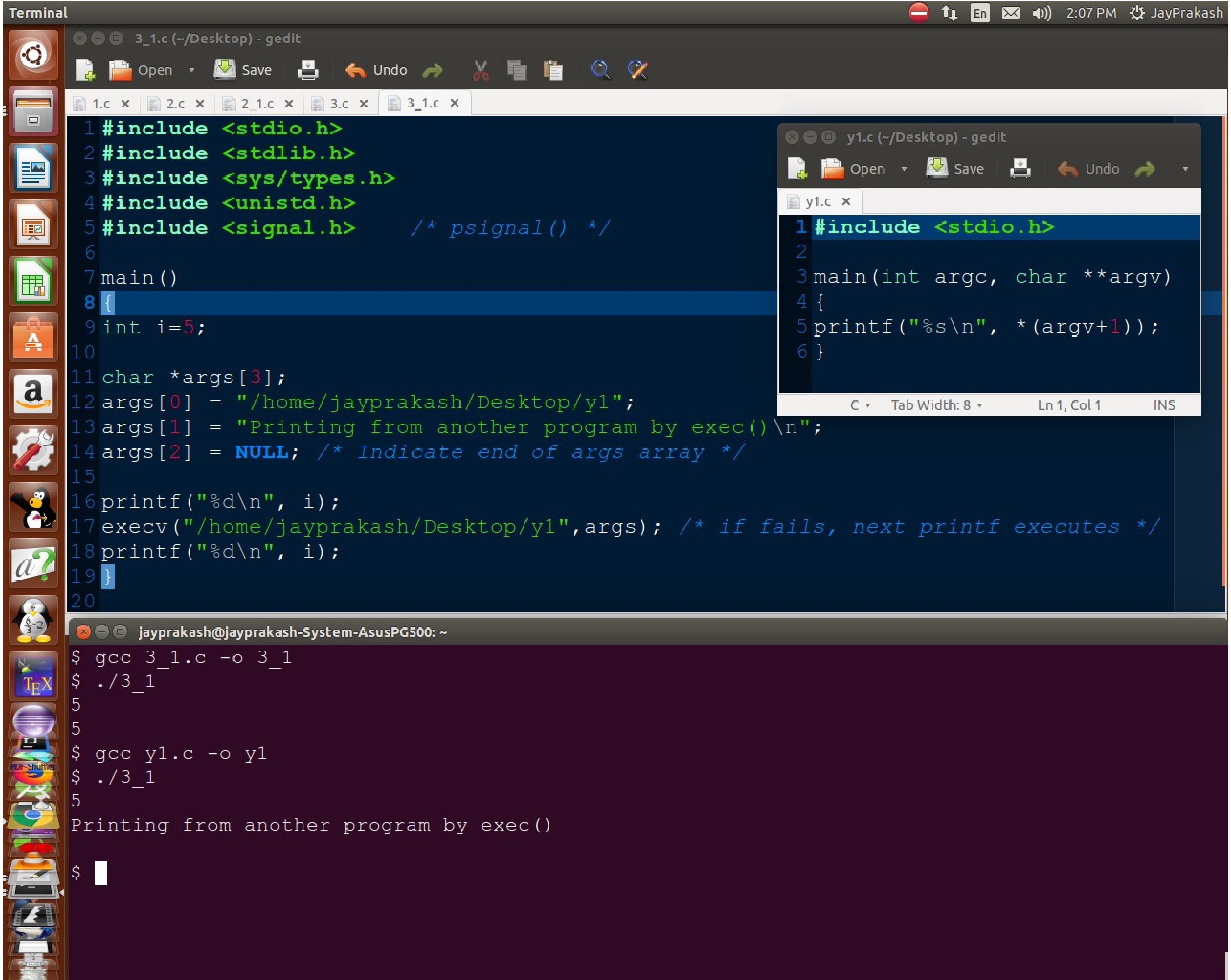
```
$ gcc 2_1.c -o 2_1
$ ./2_1
About to exec from process 6773
About to exec from process 6773
About to exec from process 6773
About to exec from process 6773
About to exec from process 6773
About to exec from process 6773
About to exec from process 6773
About to exec from process 6773
About to exec from process 6773
About to exec from process 6773
^C
$ 
```



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

main()
{
int i=5;

char *args[1];
args[0] = NULL; /* Indicate end of args array */

printf("%d\n", i);
execv("/home/jayprakash/Desktop/y",args);
printf("%d\n", i);
}
```

```
#include <stdio.h>

main()
{
printf("Printing from another program by exec()\n");
}
```

```
$ gcc 3.c -o 3.o
$ ./3.o
5
Printing from another program by exec()
$
```



```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>     /* psignal() */

main()
{
int i=5;

char *args[3];
args[0] = "/home/jayprakash/Desktop/y1";
args[1] = "Printing from another program by exec()\n";
args[2] = NULL; /* Indicate end of args array */

printf("%d\n", i);
execv("/home/jayprakash/Desktop/y1",args); /* if fails, next printf executes */
printf("%d\n", i);
}
```

```c
#include <stdio.h>

main(int argc, char **argv)
{
printf("%s\n", *(argv+1));
}
```

```
$ gcc 3_1.c -o 3_1
$ ./3_1
5
5
$ gcc y1.c -o y1
$ ./3_1
5
Printing from another program by exec()

$ 
```