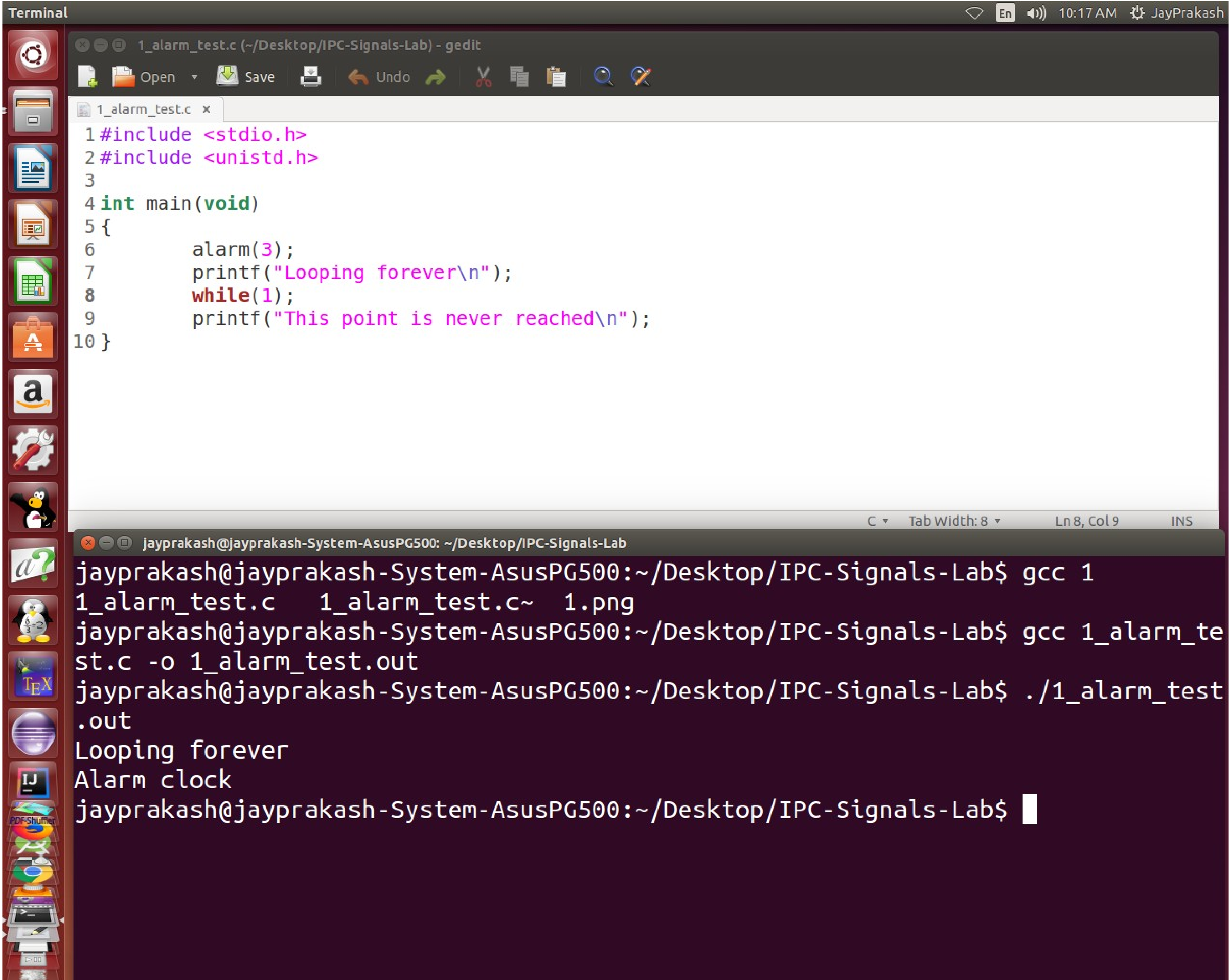
Requesting an Alarm Signal SIGALRM using alarm() System Call

System Call : unsigned int alarm(unsigned int count)

- alarm() instructs the kernel to send the SIGALRM signal to the calling process after counting seconds.
  - If an alarm had already been scheduled, that alarm is overwritten.
  - If count is 0, any pending alarm requests are cancelled.
- The default handler for this signal displays the message “Alarm clock” and terminates the process.
- alarm() returns the number of seconds that remain until the alarm signal is sent

System Call alarm() example

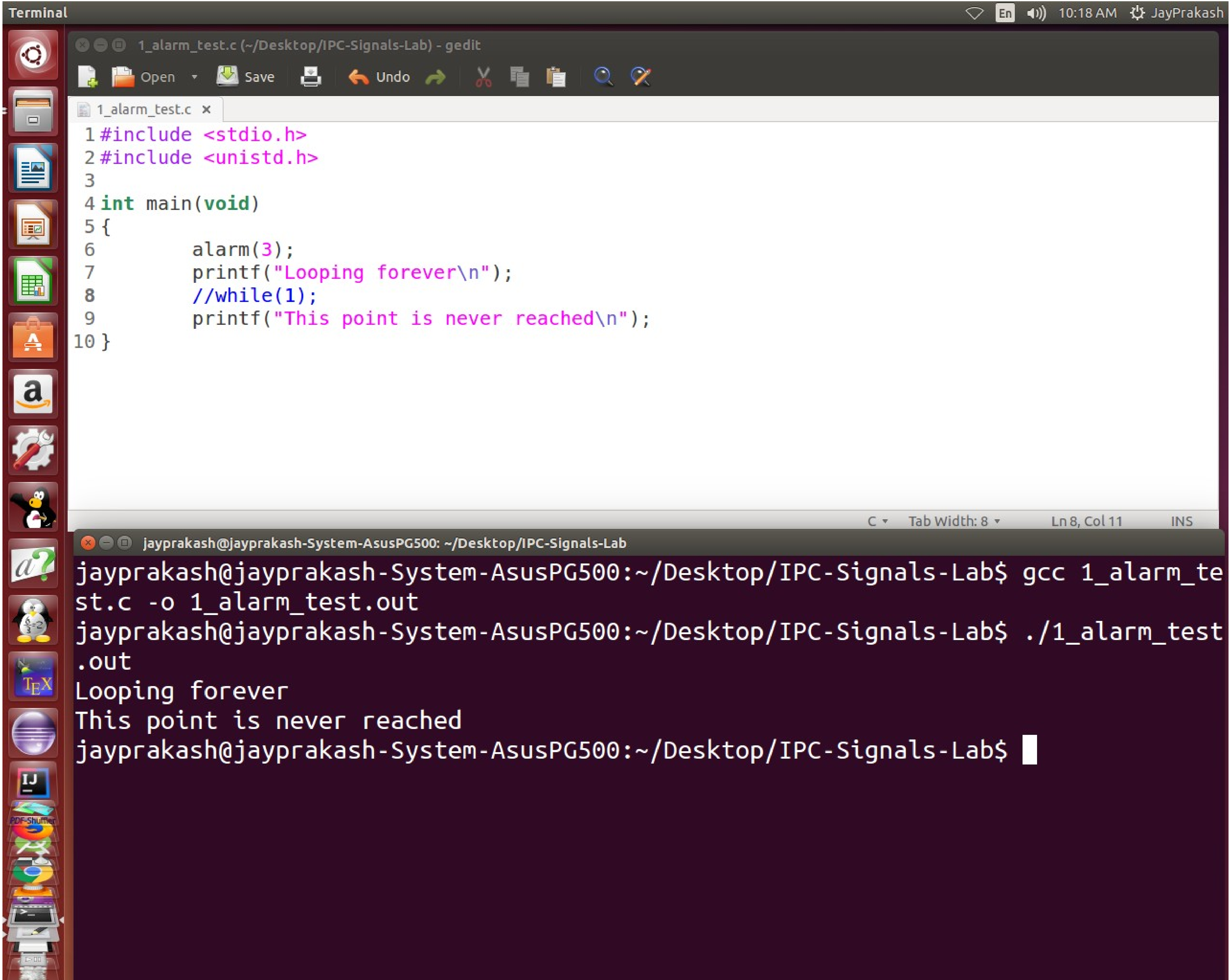
- Set Alarm for 3 second with default action handler which will display a default message “Alarm clock” and exit the program alarm_test.c



Terminal
En
10:17 AM
JayPrakash
1_alarm_test.c (~/Desktop/IPC-Signals-Lab) - gedit
Open
Save
Undo
1_alarm_test.c
C
Tab Width: 8
Ln 8, Col 9
INS
jayprakash@jayprakash-System-AsusPG500: ~/Desktop/IPC-Signals-Lab

```c
#include <stdio.h>
#include <unistd.h>

int main(void)
{
        alarm(3);
        printf("Looping forever\n");
        while(1);
        printf("This point is never reached\n");
}
```

```
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ gcc 1
1_alarm_test.c   1_alarm_test.c~  1.png
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ gcc 1_alarm_te
st.c -o 1_alarm_test.out
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ ./1_alarm_test
.out
Looping forever
Alarm clock
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ 
```


Terminal
En
10:18 AM
JayPrakash
1_alarm_test.c (~/Desktop/IPC-Signals-Lab) - gedit
Open
Save
Undo
1_alarm_test.c
C
Tab Width: 8
Ln 8, Col 11
INS
jayprakash@jayprakash-System-AsusPG500: ~/Desktop/IPC-Signals-Lab

```c
#include <stdio.h>
#include <unistd.h>

int main(void)
{
        alarm(3);
        printf("Looping forever\n");
        //while(1);
        printf("This point is never reached\n");
}
```

```
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ gcc 1_alarm_test.c -o 1_alarm_test.out
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ ./1_alarm_test.out
Looping forever
This point is never reached
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$
```

Handling Signals (Overriding Default Action)

System Call: void(*signal(int sigCode, void (*func)(int))) (int)

- signal() allows a process to specify the action that it will take when a particular signal is received.
  - sigCode specifies the signal number (as per the table shown in earlier slides) that is to be reprogrammed
  - func may be one of several values:
    - SIG_IGN, which indicates that the specified signal should be ignored and discarded
    - SIG_DFL, which indicates that the kernel's default handler should be used.
    - an address of a user-defined function, which indicates that the function should be executed when the specified signal arrives.

Handling Signals

- The valid signal numbers are stored in /usr/include/signal.h or /usr/include/bits/signum.h
  - The signals SIGKILL and SIGSTP may not be reprogrammed.
- signal() returns the previous func value associated with sigCode if successful; otherwise, it returns a value of -1
- signal() system call can be used to override the default action
- A child process inherits the signal settings from its parent during a fork()

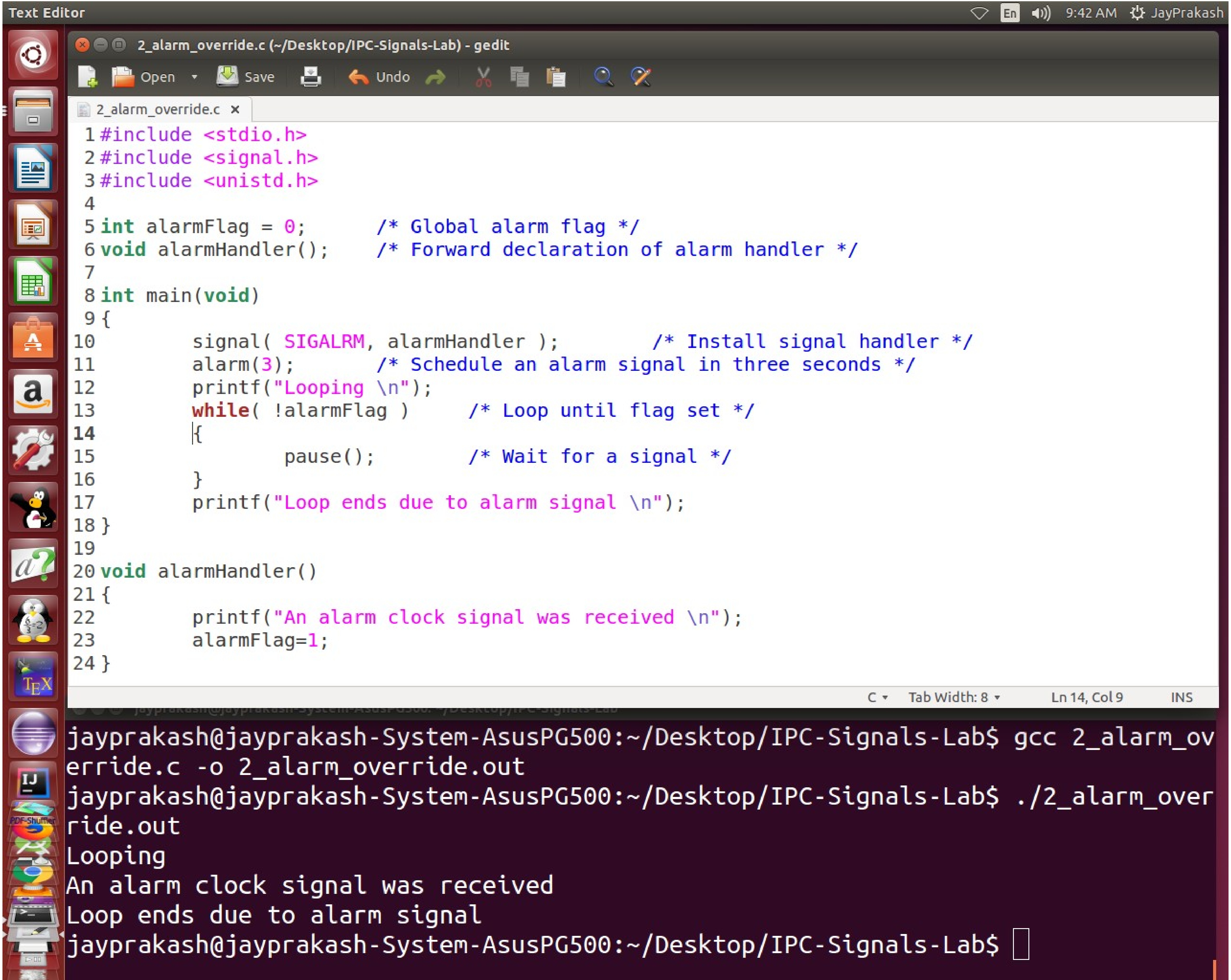
System Call pause()

System Call: int pause(void)

- pause() suspends the calling process and returns when a calling process receives a signal.
- It is most often used to wait efficiently for a signal.
- pause() doesn’t return anything useful.

Catch SIGALRM of alarm() system call using signal() system call

- Write your own handler to handle alarm signal SIGALRM
- Override default action using signal() system call
- alarm_override.c

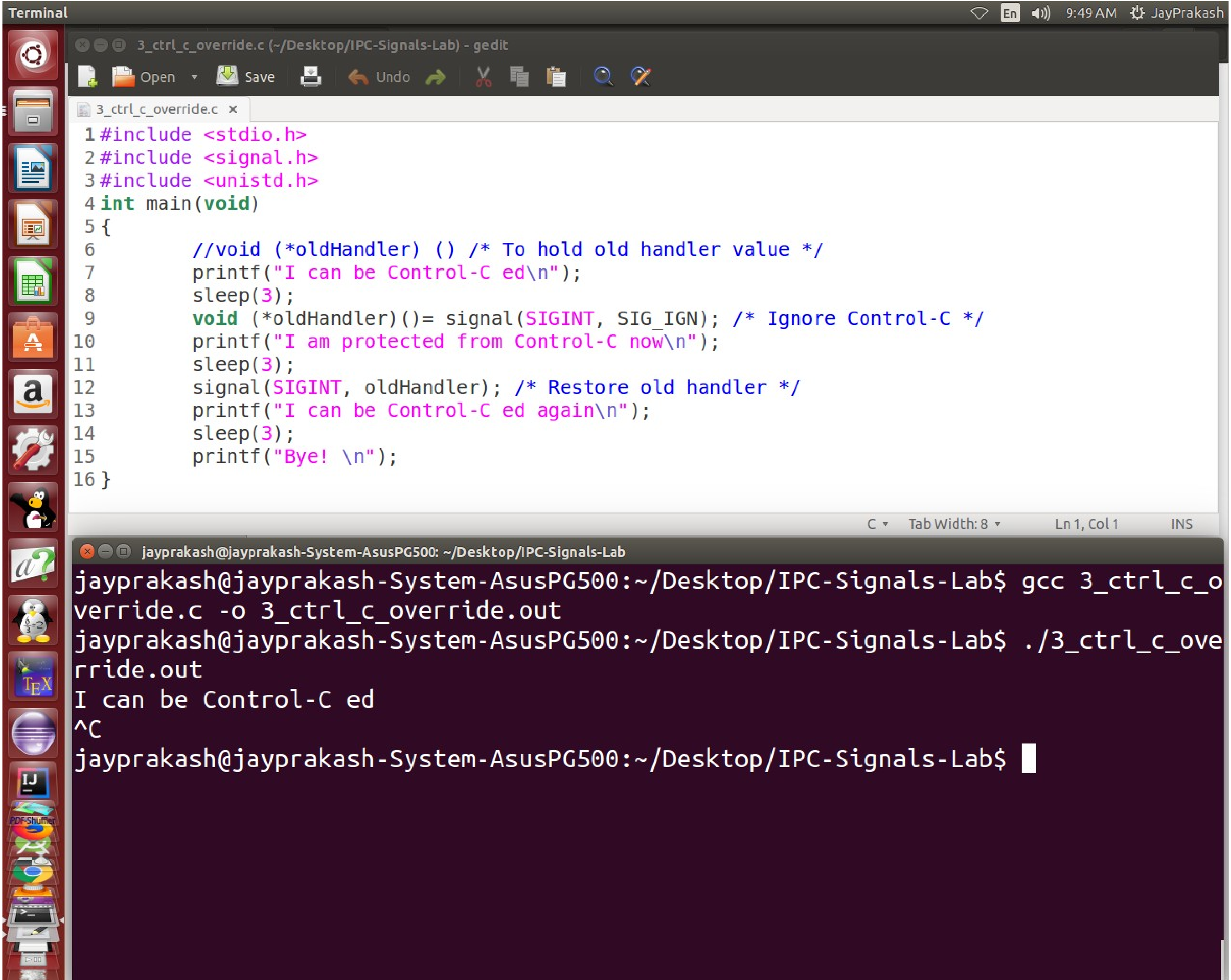


```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int alarmFlag = 0;      /* Global alarm flag */
void alarmHandler();    /* Forward declaration of alarm handler */

int main(void)
{
        signal( SIGALRM, alarmHandler );        /* Install signal handler */
        alarm(3);       /* Schedule an alarm signal in three seconds */
        printf("Looping \n");
        while( !alarmFlag )     /* Loop until flag set */
        {
                pause();        /* Wait for a signal */
        }
        printf("Loop ends due to alarm signal \n");
}

void alarmHandler()
{
        printf("An alarm clock signal was received \n");
        alarmFlag=1;
}
```

```
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ gcc 2_alarm_override.c -o 2_alarm_override.out
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ ./2_alarm_override.out
Looping
An alarm clock signal was received
Loop ends due to alarm signal
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ 
```

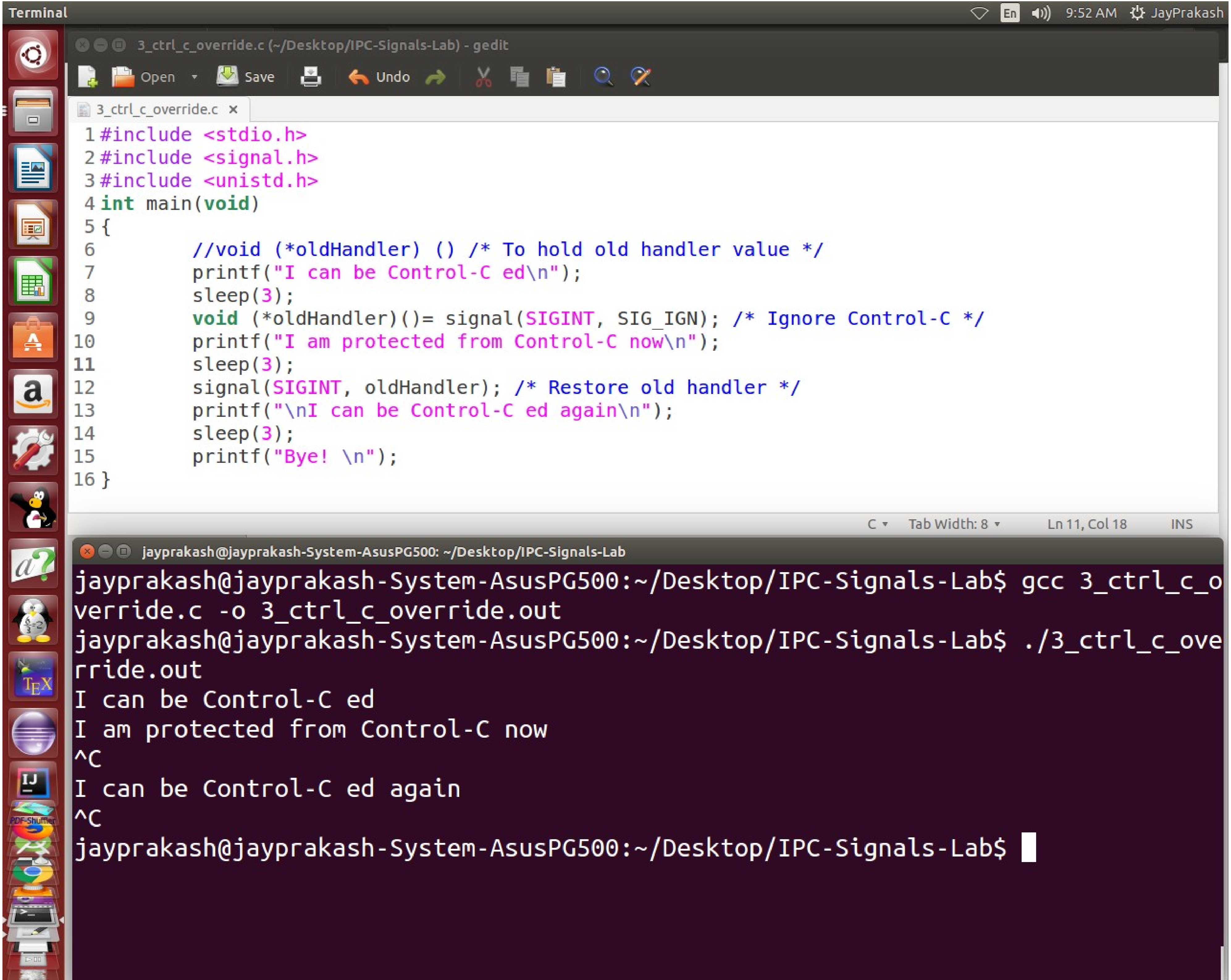
Protecting Critical Code from Ctrl-C attack

- Overriding may be used to protect critical pieces of code against Control-C attacks and other such signals.
- it can be restored after the critical code has executed.
- Here’s the source code of a program that protects itself against SIGINT signals ctrl_c_override.c


Terminal
En
9:49 AM
JayPrakash
3_ctrl_c_override.c (~/Desktop/IPC-Signals-Lab) - gedit
Open
Save
Undo
3_ctrl_c_override.c
C
Tab Width: 8
Ln 1, Col 1
INS
jayprakash@jayprakash-System-AsusPG500: ~/Desktop/IPC-Signals-Lab

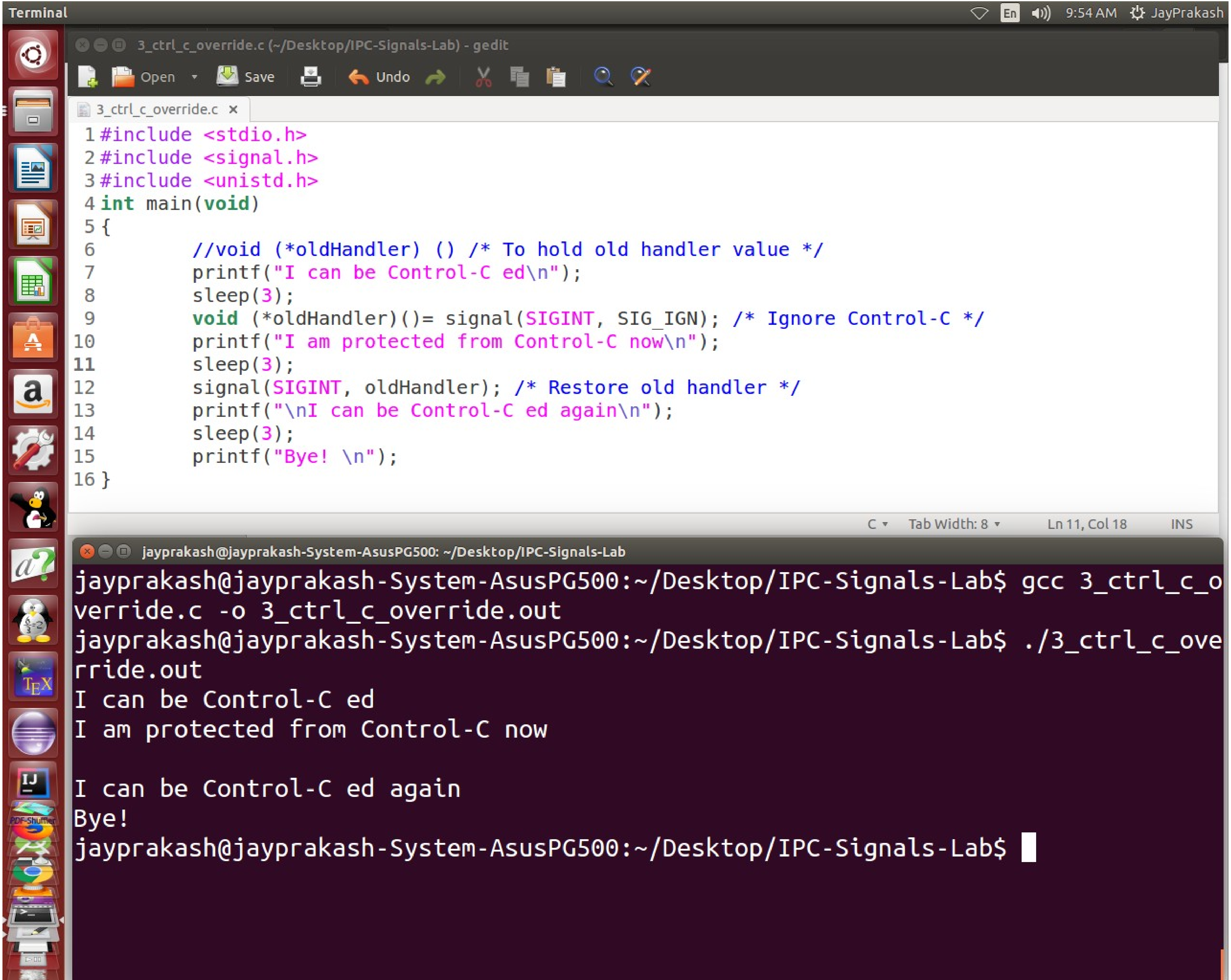
```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
int main(void)
{
        //void (*oldHandler) () /* To hold old handler value */
        printf("I can be Control-C ed\n");
        sleep(3);
        void (*oldHandler)()= signal(SIGINT, SIG_IGN); /* Ignore Control-C */
        printf("I am protected from Control-C now\n");
        sleep(3);
        signal(SIGINT, oldHandler); /* Restore old handler */
        printf("I can be Control-C ed again\n");
        sleep(3);
        printf("Bye! \n");
}
```

```
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ gcc 3_ctrl_c_override.c -o 3_ctrl_c_override.out
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ ./3_ctrl_c_override.out
I can be Control-C ed
^C
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$
```


Terminal
En
9:52 AM
JayPrakash
3_ctrl_c_override.c (~/Desktop/IPC-Signals-Lab) - gedit
Open
Save
Undo
3_ctrl_c_override.c
C
Tab Width: 8
Ln 11, Col 18
INS
jayprakash@jayprakash-System-AsusPG500: ~/Desktop/IPC-Signals-Lab

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
int main(void)
{
        //void (*oldHandler) () /* To hold old handler value */
        printf("I can be Control-C ed\n");
        sleep(3);
        void (*oldHandler)()= signal(SIGINT, SIG_IGN); /* Ignore Control-C */
        printf("I am protected from Control-C now\n");
        sleep(3);
        signal(SIGINT, oldHandler); /* Restore old handler */
        printf("\nI can be Control-C ed again\n");
        sleep(3);
        printf("Bye! \n");
}
```

```
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ gcc 3_ctrl_c_o
verride.c -o 3_ctrl_c_override.out
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ ./3_ctrl_c_ove
rride.out
I can be Control-C ed
I am protected from Control-C now
^C
I can be Control-C ed again
^C
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ 
```


Terminal
En
9:54 AM
JayPrakash
3_ctrl_c_override.c (~/Desktop/IPC-Signals-Lab) - gedit
Open
Save
Undo
3_ctrl_c_override.c
C
Tab Width: 8
Ln 11, Col 18
INS
jayprakash@jayprakash-System-AsusPG500: ~/Desktop/IPC-Signals-Lab

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
int main(void)
{
        //void (*oldHandler) () /* To hold old handler value */
        printf("I can be Control-C ed\n");
        sleep(3);
        void (*oldHandler)()= signal(SIGINT, SIG_IGN); /* Ignore Control-C */
        printf("I am protected from Control-C now\n");
        sleep(3);
        signal(SIGINT, oldHandler); /* Restore old handler */
        printf("\nI can be Control-C ed again\n");
        sleep(3);
        printf("Bye! \n");
}
```

```
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ gcc 3_ctrl_c_o
verride.c -o 3_ctrl_c_override.out
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ ./3_ctrl_c_ove
rride.out
I can be Control-C ed
I am protected from Control-C now

I can be Control-C ed again
Bye!
jayprakash@jayprakash-System-AsusPG500:~/Desktop/IPC-Signals-Lab$ 
```

Process Groups and Control Terminals

- When you’re in a shell and you execute a program that creates several children, a single Control-C from the keyboard will normally terminate the program and its children and then return you to the shell.
- In order to support this kind of behavior, UNIX introduced a few new concepts.
    - In addition to having a unique process ID number, every process is also a member of a process group.
        - Several processes can be members of the same process group.
        - When a process forks, the child inherits its process group from its parent.
        - A process may change its process group to a new value by using setpgid()
        - When a process execs, its process group remains the same.

Process Groups and Control Terminals

- Every process can have an associated control terminal, which is typically the terminal where the process was started.
  - When a process forks, the child inherits its control terminal from its parent.
  - When a process execs, its control terminal stays the same.
- Every terminal can be associated with a single control process.
  - When a metacharacter such as a Control-C is detected, the terminal sends the appropriate signal to all of the processes in the process group of its control process.
- If a process attempts to read from its control terminal and is not a member of the same process group as the terminal’s control process,
  - the process is sent a SIGTTIN signal, which normally suspends the process.

Process Groups and Control Terminals

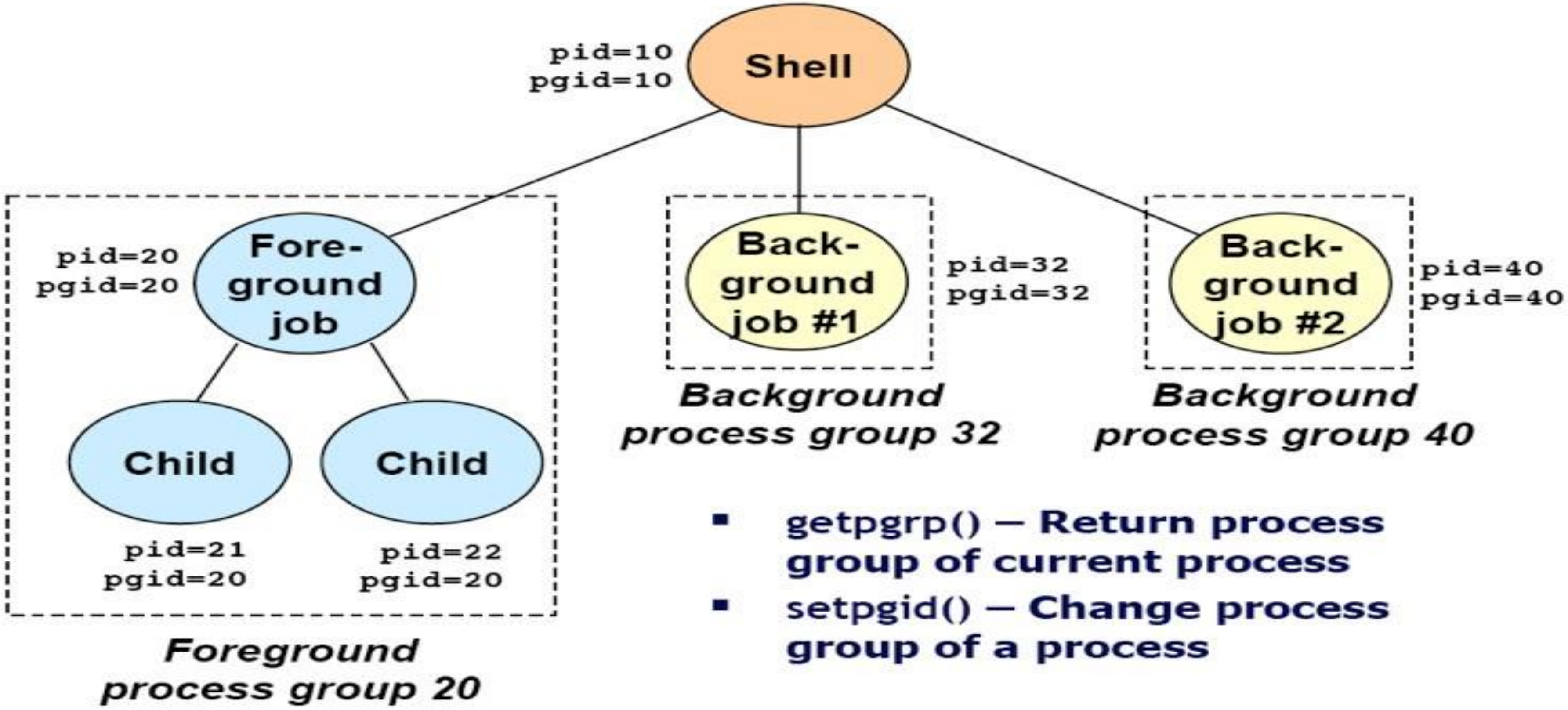
- When an interactive shell begins, it is the control process of a terminal and has that terminal as its control terminal.
- When a shell executes a foreground process:
  - the child shell places itself in a different process group before exec’ing the command and takes control of the terminal.
  - Any signals generated from the terminal thus go to the foreground command rather than to the original parent shell.
  - When the foreground command terminates, the original parent shell takes back control of the terminal.

Process Groups and Control Terminals

- When a shell executes a background process:
  - the child shell places itself in a different process group before executing, but does not take control of the terminal.
  - Any signals generated from the terminal continue to go to the shell.
  - If the background process tries to read from its control terminal, it is suspended by a SIGTTIN signal.

Process Groups and Control Terminals

- **Every process belongs to exactly one process group.**



- **getpgrp() – Return process group of current process**
- **setpgid() – Change process group of a process**