Part 2:

4.17 The program shown in Figure 4.16 uses the Pthreads API. What would be the output from the program at LINE C and LINE P?

```
#include <pthread.h>
#include <stdio.h>
#include <types.h>

/* the thread */
int value = 0;
void *runner(void *param);
int main(int argc, char *argv[])
{
      pid_t pid;
      pthread_t tid;
      pthread_attr_t attr;
      pid = fork();

      /* child process */
      if (pid == 0) {
            pthread_attr_init(&attr);
            pthread_create(&tid,&attr,runner,NULL);
            pthread_join(tid,NULL);
            printf("CHILD: value = %d",value); /* LINE C */
      }
      /* parent process */
      else if (pid > 0) {
            wait(NULL);
            printf("PARENT: value = %d",value); /* LINE P */
      }
}

void *runner(void *param)
{
      value = 5;
      pthread exit(0);
}
```

ANSWER:

LINE C = 5

LINE P = 0

Part 3:
Read and summarize following page:
http://www.thegeekstuff.com/2013/11/linux-process-and-thread

Summary:

Processes are fundamental to Linux as every work done by the OS is done in terms of and by the processes. Processes have priority based on which kernel context switches them. A process can be pre-empted if a process with higher priority is ready to be executed.

Processes can talk to other processes using Inter process communication methods and can share data using techniques like shared memory.

Threads in Linux are nothing but a flow of execution of the process. A process containing multiple execution flows is known as multi-threaded process. Effectively we can say that threads and light weight processes are same. It's just that thread is a term that is used at user level while light weight process is a term used at kernel level.

The main difference between a light weight process (LWP) and a normal process is that LWPs share same address space and other resources like open files etc. As some resources are shared so these processes are considered to be light weight as compared to other normal processes and hence the name light weight processes.

RUNNING – This state specifies that the process is either in execution or waiting to get executed.
INTERRUPTIBLE – This state specifies that the process is waiting to get interrupted as it is in sleep mode and waiting for some action to happen that can wake this process up. The action can be a hardware interrupt, signal etc.

UN-INTERRUPTIBLE – It is just like the INTERRUPTIBLE state, the only difference being that a process in this state cannot be waken up by delivering a signal.

STOPPED – This state specifies that the process has been stopped. This may happen if a signal like SIGSTOP, SIGTTIN etc is delivered to the process.

TRACED – This state specifies that the process is being debugged. Whenever the process is stopped by debugger (to help user debug the code) the process enters this state.

ZOMBIE – This state specifies that the process is terminated but still hanging around in kernel process table because the parent of this process has still not fetched the termination status of this process. Parent uses wait() family of functions to fetch the termination status.

DEAD – This state specifies that the process is terminated and entry is removed from process table. This state is achieved when the parent successfully fetches the termination status as explained in ZOMBIE state.