Some solutions to exercises on Threads

1. Does the multithreaded web server below exhibit task or data parallelism?

   sol: Data parallelism is usually associated with applications where we have a huge data structure where the same computation (same function) is applied in parallel to different elements in the data structure. The picture below does not seem to correspond to such application. However, if the server always call the same function to answer requests, then we might be correct to call this example data parallelism because it always execute the same function on different requests. If the server calls different functions, then it is definitely task parallelism.

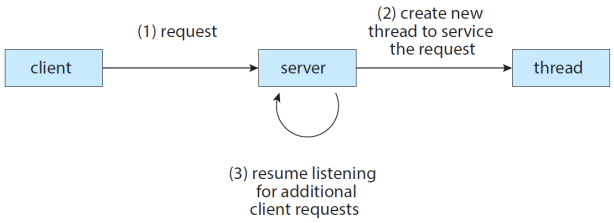

Figure 4.2 Multithreaded server architecture.

2. Does every process has a kernel thread?

   sol: The answer is yes. The kernel scheduler only schedule kernel thread, it does not know about user level threads. Thus, to execute code, even if the code is managed by a user level thread library, there must be a kernel thread for the code to be executed, the user level thread must be mapped to a kernel level thread as we already know it.

3. Which of the following components of program state are shared across threads in a multithreaded process? 1- Register values; 2- Heap memory; 3- Static variables; 4- Stack memory

4. Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain

   sol: If user level threads are all scheduled to the same kernel thread then definitely no. If user level threads are mapped using a one-to-one mapping to kernel threads, then yes.

5. Is it possible to have concurrency but not parallelism? Explain

   sol: Yes. If the computer cheap has only one cpu, then hardware parallelism is not possible, so only concurrency. Even if multithreading is available at user or kernel level, threads of a same application can only be executed sequentially, thus the many threads created by a process will only execute concurrently.

6. Consider the following code segment:

```
pid t pid;
pid = fork();
```

```
if (pid == 0) { /* child process */
fork();
thread create( . . .);
}
fork();
```

(a) How many unique processes are created?

(b) How many unique threads are created?

7. Consider the program on the next page, what would be the output from the program at LINE C and LINE P?

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

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pid_t pid;
pthread_t tid;
pthread_attr_t attr;

   pid = fork();

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

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

8. Provide two programming examples of multithreading giving improved performance over a single-threaded solution.

   Sol: 1- A Web server that services each request in a separate thread. 2- A parallelized application such as matrix multiplication where different parts of the matrix may be worked on in parallel.

9. What are two differences in terms of context switch and scheduling between user-level threads and kernel-level threads?

   Sol: User-level threads are much faster to switch between, as there is no context switch; further, a problem-domain-dependent algorithm can be used to schedule among them. ... Kernel-level threads are scheduled by the OS, and each thread can be granted its own timeslices by the scheduling algorithm.

10. Describe some of the actions taken by a kernel to context switch 1- Among threads; 2- Among processes

    Sol: 1- To context switch between threads, the kernel only needs to switch the context within the process. Swapping between threads, saves the state into tcp (thread control block), then load from the next one from the scheduler. 2- Save the state of the old process into the PCB (process control block) and load the saved state for the new process from the short term scheduler.

11. Describe some of the actions taken by a kernel to context-switch between kernel level threads

    Sol: Context switching between kernel threads typically requires saving the value of the CPU registers from the thread being switched out and restoring the CPU registers of the new thread being scheduled.

12. What are some of the resources used when a thread is created? How do they differ from those used when a process is created?

    Sol: Because a thread is smaller than a process, thread creation typically uses less resources than process creation. Creating a process requires allocating a process control block (PCB), a rather large data structure. The PCB includes a memory map, list of open files, and environment variables. Allocating and managing the memory map is typically the most time-consuming activity. Creating either a user or kernel thread involves allocating a small data structure to hold a register set, stack, and priority.