Lecture

- In the lecture on March 2nd we will finish ythe discussion on Chapter 3. Examples will be shown for implementation and usage of ordinary pipes and named pipes. Examples will be shown for implementation and usage of socket programming.
- We will start with Chapter 4 (Multithreaded Programming). Next week we will start with Chapter 5 (Process Scheduling).
- For Chapters 3 and 4 examples were be shown illustrating the difference of fork, vfork, and clone on Linux systems.
- For Chapters 3 and 4 axamples will be shown of how to implement signal handlers. Examples will be shown of how to use OpenMP and pthreads.
- Note, that you find even more exercises including solutions here :
  `http://codex.cs.yale.edu/avi/os-book/OS9/practice-exer-dir/index.html`
- Prepare for the Tutorial Session in week 10, 2018.:
  All exercises not discussed so far. In addition:

  4.1 Provide two programming examples in which multithreading does not provide better performance than a single-threaded Solution.

  4.2 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

  4.3 Which of the following components of program state are shared across threads in a multithreaded process?

  * Register values
  * Heap memory
  * Global variables
  * Stack memory

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

  4.5 In Chapter 3, we discussed Google's Chrome browser and its practice of opening each new website in a separate process. Would the same benefits have been achieved if instead Chrome had been designed to open each new website in a separate thread? Explain.

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

  4.7 Using Amdahl's Law, calculate the speedup gain of an application that has a 60 percent parallel component for (a) two processing cores and (b) four processing cores.

4.9 A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between startup and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).

* How many threads will you create to perform the input and output? Explain.
* How many threads will you create for the CPU-intensive portion of the application? Explain.

4.10 Consider the following code segment:

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

How many unique processes are created? How many unique threads are created?

4.11 As described in Section 4.7.2, Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way, allowing a task to be more akin to a process or a thread depending on the set of flags passed to the clone() system call. However, many operating systems, such as Windows XP and Solaris, treat processes and threads differently. Typically, such systems use a notation wherein the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.

4.13 Consider a multiprocessor system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be more than the number of processors in the system. Discuss the performance implications of the following scenarios.

* The number of kernel threads allocated to the program is less than the number of processors.
* The number of kernel threads allocated to the program is equal to the number of processors.
* The number of kernel threads allocated to the program is greater than the number of processors but less than the number of user-level threads.

4.14 Pthreads provides an API for managing thread cancellation. The `pthread_setcancelstate()` function is used to set the cancellation state. Its prototype appears as follows:

`pthread_setcancelstate(int state, int *oldstate)` The two possible values for the state are `PTHREAD_CANCEL_ENABLE` and `PTHREAD_CANCEL_DISABLE`. Using the code segment shown in Figure below, provide examples of two operations that would be suitable to perform between the calls to disable and enable thread cancellation.

```
int oldstate;

pthread setcancelstate(PTHREAD CANCEL DISABLE, &oldstate);
/* What operations would be performed here? */
pthread setcancelstate(PTHREAD CANCEL ENABLE, &oldstate);
```

4.17 The program shown in Figure below 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>
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);
}
```