COS Questions – Lecture 2

Operating System Concepts (Tenth Edition)

Threads & Concurrency

4.1 Provide three programming examples in which multithreading provides better performance than a single-threaded solution.

In general, any problem that can be sub-divided into a number of tasks, will benefit from parallelization. Examples include:

- A web server that services each request in a separate thread
- Matrix multiplication
- A Divide-and-Conquer sorting algorithm (example)

4.2 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.

Using MATLAB (because I'm lazy):

```
amdahl = @(s,n) 1/(s + ((1-s)/n));

amdahl(0.4, 2) % -> 142.9 percent
amdahl(0.4, 4) % -> 181.8 percent
```

4.3 What are two key differences between user-level threads and kernel-level threads?

- User-level threads are unknown by the kernel, whereas the kernel is aware of kernel threads.
- On systems using either many-to-one or many-to-many model mapping, user threads are scheduled by the thread library, and the kernel schedules kernel threads.
- Kernel threads need not be associated with a process, whereas every user thread belongs to a process. Kernel threads are generally more expensive to maintain than user threads, as they must be represented with a kernel data structure.

A good summary of the differences (source):

User-Level Threads	Kernel-Level Thread
User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

4.4 Which of the following are shared across threads in a multithreaded process?

a. Register values: No.
b. Heap memory: No.
c. Global variables: Typically, yes (depends on the language/platform).
d. Stack memory: No.

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

Yes, it is possible to have concurrency but not parallelism.

Concurrency means where two different tasks (threads) start working together in an overlapped time period, however, it does not mean they run at same instant.

Parallelism is where two or more different tasks start their execution at the same time. It means that the two tasks start working simultaneously, e.g., on a multi-processing system with multiple cores.

4.6 Determine if the following problems exhibit task or data parallelism.

They are defined as:

Data Parallelism Distributing subsets of the same data across multiple computing cores and performing the same operation on each core.	**Task Parallelism** Distributing tasks (threads) across multiple computing cores. Each thread is performing a unique operation.
Example: Splitting up an array of integers to compute the individual sums, then computing the final sum by adding the individual components.	**Example:** A word processor might have a spell-checker thread and an I/O thread, both doing different tasks.

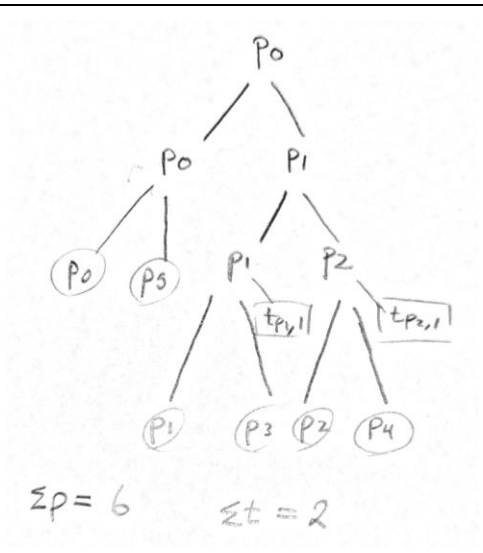
Therefore:

Using a separate thread to generate a thumbnail for each photo in a collection.	Task Parallelism
Transposing a matrix in parallel.	Data Parallelism
A networked application where one thread reads from the network and another thread writes to the network.	Task Parallelism

4.7 Given the following code segment, how many unique processes are created? How many unique threads are created?

```
pid_t pid;

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



The diagram above shows that a total of 6 processes and 2 threads were created.

What should have been made clear is that each newly created thread starts a different function than the one currently executing, defined by one of the arguments to a `thread_create()`, as a pointer to the function to begin executing. The threads therefore don't move on to the last `fork()` call.

CPU Scheduling

5.1 What is the difference between an CPU burst and I/O burst?

Most programs have some alternating cycle of CPU computation (CPU Burst) and waiting for I/O of some kind (I/O Burst), where on simple system running a single process, the time spent waiting for I/O are wasted CPU cycles.

5.2 What is the role of a CPU scheduler? Why is it necessary?

A CPU scheduler manages how processes share available computation resources via a scheduling algorithm, the challenge of which is to make the overall system as "efficient" and "fair" as possible.

5.3 What is the difference between pre-emptive and non-pre-emptive scheduling?

A pre-emptive scheduler is able to interrupt a process, where as non-pre-emptive scheduling relies on the process giving up control voluntarily.

5.4 What is the role of the dispatcher? What is the difference between a voluntary and non-voluntary context switch?

The dispatcher is a component of a CPU scheduler which manages the actual dispatching of processes, i.e., the context switches, switching to user mode and resuming a program. A voluntary context switch involves a process giving up control (e.g. when waiting for an I/O resource), whereas a non-voluntary context switch is a process getting pre-empted.

5.5 What are the five criteria (parameters) for evaluation of CPU scheduling algorithms?

CPU Utilization Highest possible utilization of the CPU's available resources.	**Throughput** Measure of number of processes executed per time unit.	**Turnaround Time** Sum of waiting times from all queues and computation times.
Waiting Time Sum of waiting times of processes in the ready queue.	**Response Time** The time between a request and response.	**Variance*** The variance of response time, desired to be kept low.

5.6 What is starvation? Which of the following algorithms could result in starvation? How can starvation be resolved?

Starvation is when a process never gets executed, because processes of higher priority always get executed before it.

a. First-come, first-served
b. Shortest job first
c. Round robin
d. Priority

Starvation can be resolved by **aging**, where starved processes steadily increase in priority.

5.7. What advantage is there in having different time-quantum sizes at different levels of a multilevel queueing system?

Processes that need more frequent servicing – e.g., interactive processes such as editors – can be in a queue with a small time quantum. Processes with no need for frequent servicing can be in a queue with a larger quantum, requiring fewer context switches to complete the processing and thus making more efficient use of the computer.

5.8 What is the difference between symmetric multiprocessing (SMP) and asymmetric multiprocessing?

Asymmetric Multiprocessing	**Symmetric Multiprocessing (SMP)**
One processor is the master, controlling all activities and running all kernel code, while the other cores only execute user code.	Each processor schedules its own processes, either from a common ready queue or from separate ready queues for each processor.

5.9 How can threads (ready queues) be organized on multiprocessing processors?

1. All threads may be in a common ready queue.
2. Each processor may have its own private queue of threads (typically used).

5.10 What is memory stall? How do modern processors resolve this issue?

When a processor accesses memory, it spends a significant amount of time waiting for the data to become available. Modern processor remedy this by hardware multithreading. That way, if one hardware thread stalls while waiting for memory, the core can switch to another thread.

5.11 What is load balancing? What are the two general approaches to achieve it?

Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system. This can be done via:

Push Migration	**Pull Migration**
Involves a separate process that runs periodically, (e.g. every 200 ms), and moves processes from heavily loaded processors onto less loaded ones.	Involves idle processors taking processes from the ready queues of other processors.

5.12 What is processor affinity? What are the two types?

SMP systems attempting to keep processes on the same processor is known as processor affinity, differing between soft (no-guarantee) or hard affinity (strict).

5.13 What are the two types of real-time scheduling? How are they different?

Soft Realtime System	**Hard Realtime System**
Resulting in degraded performance if a deadline cannot be met, e.g. video streaming.	Resulting in total failure if a deadline cannot be met, e.g. automobile air-bag deployment.

5.14 What is event latency? What two types of latencies affect the performance of real-time systems?

Event latency is the amount of time that elapses from when an event occurs to when it is serviced by a real-time system. The two types of latencies are

1. **Interrupt latency**
   Period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt.
2. **Dispatch latency**
   The amount of time required for the scheduling dispatcher to stop one process and start another.