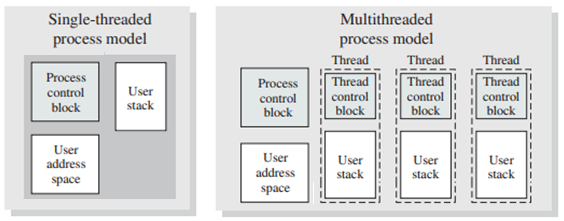
# Section 4: Threads

## Question 1



Using this figure, contrast the structure of the Single-Threaded Process (STP) model with the Multi-Threaded Process (MTP) model.

1. Describe differences between the Process Control Block in the STP and PCB / TCB in the MTP.
2. Describe the differences in how user stacks are maintained in each model.
3. Describe the differences in how processor state information (i.e. program counter, register contents, etc.) is maintained in each model.
4. Describe how OS allocated resources (files, semaphores, sockets, etc.) are managed in each model.

### Answer

1. In both models, a “process” is an address space maintaining the process image. The image contains the user address space (instructions and data (heap)) and control stack. In the single-threaded model, each process has only a single thread of execution and so there is little distinction between the process and its thread e.g. both process and single thread share a single process control block. In the multithreaded model, the process provides a context in which one or more threads execute.
2. In the STP model, there exist only a single thread so only a single user stack per process is needed. The MTP model has multiple threads each of which maintains its own instruction trace (thread of control) and each thread requires a separate user stack.
3. In the STP model there is only a single thread whose processor state information must be saved and restored when the process is context switched or interrupted. So in the STP, processor state can be maintained in the Process Control Block.   
   In the MTP model, each thread maintains its own execution state. Each thread is individually context switched and interrupted so each thread’s processor state is maintained in its Thread Control Block.
4. In the MTP model, all threads share access to the resources allocated to the process as a whole. These resources are maintained in the single Process Control Block maintained by each process and shared by all threads.

## Question 2

How can the use of threads increase the performance of an application when run on a multiprocessor? Use the example of Worker Threads pattern to illustrate your answer.

### Answer

Each thread represents a ‘thread of execution’ in the program. When executed on a multiprocessor, each thread can execute on its own processor in parallel with other threads increasing the overall processing throughput of the system.

In the case of the worker threads pattern, if each processor is capable of processing N work items per second, then M processors can process M\*N work items per second (in theory anyway).

## Question 3

What are the three advantages provided by Threads over Processes given in the slides?

### Answer

1. Threads can be created and destroyed more efficiently than processes.
2. N threads require fewer resources than the same number of processes. This means we can support N threads of execution in an application without the overhead of N processes.
3. Threads simplify the communication between concurrently executing instruction traces. Threads share access to the memory, files, sockets, and other resources allocated to their process while processes are isolated from each other.

## Question 4

What is the significance of blocking I/O (e.g. a network socket read) in the single vs. multi-threaded models? Use the example of the monitoring application presented in the slides.

### Answer

When programming with a single-thread, blocking I/O will halt the execution of the entire process. If the monitoring application were implemented using a single thread, that thread would be blocked waiting for an incoming message and would freeze the GUI i.e. the GUI would not update or respond to user input while the single thread blocks waiting for a new message.

When programming with multiple threads, we design a process with threads that serve specific purposes. We can create a thread that blocks waiting for network messages and a second thread that maintains the GUI’s presentation.

## Question 5

Describe the three multithreading control strategies presented in the slides.

### Answer

**Note**: For the exam, pay attention to the pseudo code associated with each answer.

Worker Threads: If the problem involves processing individual units of work, multiple threads can be employed to process several work units in parallel (assuming a multiprocessor). See Slides.

while (true) {

workItem = FIFO.take();

process(workItem);

}

Task Scheduling: One or more tasks (i.e. some action that is required to maintain the system) are periodically scheduled (e.g. once every N seconds) and executed. A thread can be created which is programmed to alternatively sleep for N seconds (the wait period between task executions) and execute the task after the sleep period.

while (true) {

sleep(N);

doTask()

}

Event Handling: Events are delivered to the system from many sources, each of which requires a specific response from the system. Each event handler can be executed in own thread.

while (true) {

message = socket.read(); // Blocking I/O Read

process(message);

}

Note in this example the socket.read() operation is blocking i.e. the thread’s execution will block until a message arrives on the socket.

## Question 6

Describe the significance of Figure 4.3 A and B in terms of the time needed to execute the two RPC client calls with and without threads? Hint: Describe in terms of the time the process spends blocked waiting for the response from the remote RPC server.

Is a multi-core multiprocessor needed to obtain this increase in performance?

### Answer

Figure 4.3a illustrates that without multiple threads, two blocking RPC calls must be executed sequentially taking 2x to execute both requests. Figure 4.3b illustrates that when we execute each blocking I/O request in a separate thread, we can see an overall improvement in the system’s performance. The second request can be executed in Thread 2 while the first request blocks the execution of thread one allowing both requests to be executed concurrently.

Most interesting is that the increase in performance is obtained using a uniprocessor (NOT SMP). While thread one’s execution is blocked waiting for the server’s response, the second RPC call can be made in a second thread on the same (single) processor.

## Question 7

What does Amdahl’s Law tell us about scaling up an application’s performance using multiple processors (SMP)?

Using the speedup formula (section 4.3), how much of a speedup can we expect to see from a program containing 10% serial code executed on 9 processors?

### Answer

Ideally we would like to obtain a linear speedup in the performance of our programs by splitting their processing across multiple processors i.e. a 4x speedup in performance when running on a four core multiprocessor.

However, most program designs have two types of processing: The processing that can be run in parallel and that which must be executed sequentially i.e. *sections* of instructions that cannot be executed in parallel.

Amdahl’s law tells us that even a relatively small percentage of sequential processing significantly limits the effects of scaling the program’s performance across multiple processors.

If we are able to scale 90% of the program’s execution, we get a speedup of = 5.

## Question 8

Describe the implementation of Threads in the Linux kernel as described in Chapter 3 of the book Linux Kernel Development in the Supplemental Materials folder.

How does the approach Linux takes differ from Windows?

### Answer

Linux implements threads using its process mechanisms. That is, a Linux thread is implemented using the same structures as a ‘normal’ process and there is no significant difference between a Linux Process and a Linux Thread. The major difference is that thread-processes share the same image and other resources while normal processes are isolated from each other.

Other OS (e.g. Windows and Solaris), Processes and Threads are distinctly different objects / resources. Each has its own specific kernel data structures and scheduling policies.

## Question 9 (20 Points)

Perform your own research and describe the Thread Pool design pattern. Provide an example of an application of the thread pool pattern.

How does this pattern handle the creation and management of threads? Hint: What is meant by ‘pool’?

How does the pattern limit the number of executing threads when the number of tasks / work items grows very large?

### Answer

A thread pool is a design pattern describing a mechanism which uses threads to execute tasks. A task represents a ‘unit of work’ that needs to be processed. Tasks are dispatched to available threads that provide a context for the task’s execution. A common example is a network server that receives requests from remote clients i.e. a HTTP server. Each of the client’s requests is converted into a task that is executed by a separate thread.

Although when compared to processes, the creation of threads is relatively inexpensive, system resources are consumed each time a thread is created and destroyed. Instead of creating and destroying a thread for each task’s execution, the thread pool maintains a set (pool) of threads that can be dispatched when needed to execute a task. The size of the pool is limited so as to keep too many threads from being created during the busy hour.

At times (e.g. during busy hour) the number of requests / tasks may exceed the number of threads that can be supported by the hosting system. To address this issue, the pool size is limited to a maximum number of threads and tasks awaiting execution are maintained in a queue.