2.6 Threads

The thread concept

An application implemented as a single process follows a single path of execution through the program. Whenever the execution blocks, waiting for some resource to become available, the entire process blocks. Many applications have parts that could run concurrently if only each could block independently. Similarly, independent parts could run in parallel on multiple CPUs.

Creating a separate process for each of potentially many short-lived activities is too inefficient. A ***thread*** is an instance of executing a portion of a program within a process without incurring the overhead of creating and managing separate PCBs.

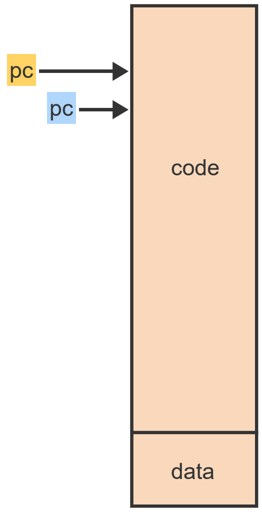
Example 2.6.1: Typical uses of threads.

- A browser can draw an image on the screen while at the same time waiting for data from the Internet. One thread can be busy drawing the image while another can keep blocking while retrieving the data.
- A word processor can run a spell checker at the same time as the user is typing. Waiting for each typed character and displaying the data on the screen can be done concurrently by two separate threads.
- An Internet server, such as an email or web page server, is receiving many requests from different users at the same time. When each request is implemented as a new thread, many requests can proceed concurrently without holding up the progress of others.

Feedback?

PARTICIPATION ACTIVITY | 2.6.1: Multi-threaded execution of the same code.

Start 2x speed



Captions

1. A single-threaded process has a single program counter (pc).
2. The pc moves through the program as execution progresses.
3. A process with multiple threads has a separate pc for each thread.

4. The different pc's advance through the code independently from one another as each thread executes a different sequence of instructions within the same code.
5. When multiple CPUs are available, the threads can proceed in parallel.

Feedback?

PARTICIPATION ACTIVITY | 2.6.2: Threads within a process.

1) Multiple threads within a process _____.

- will always execute the same sequence of instructions
- must execute within disjoint parts of the shared program
- (selected) may execute different sequences of instructions within any part of the shared program

Correct

Each thread is an independent entity and can execute any sequence of instruction within the program regardless of any other thread.

Feedback?

PARTICIPATION ACTIVITY | 2.6.3: Threads vs processes.

Using n threads within a single process is more efficient than using n separate processes because _____.

1) the threads share the same code and data

- (selected) True
- False

Correct

The code and data do not have to be replicated n times.

2) only the threads can take advantage of multiple CPUs

- True
- (selected) False

Correct

Both processes and threads can take advantage of multiple CPUs.

3) only threads can block independently from one another

- True
- (selected) False

Correct

Both processes and threads can block independently from one another.

Feedback?

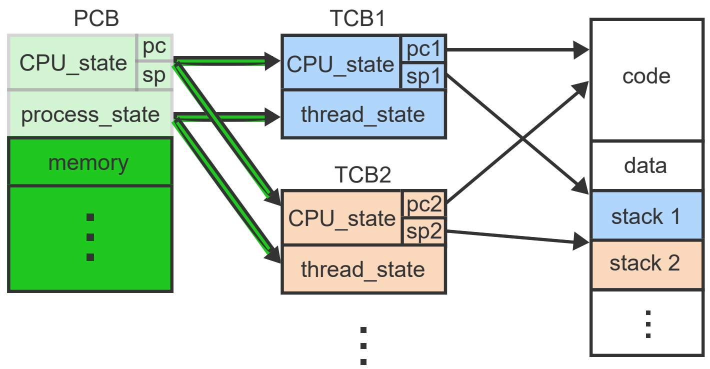
The thread control block

Since each thread within a process is an independent execution activity, the runtime information held in the PCB and the execution stack must be replicated for each thread. A ***thread control block (TCB)*** is a data structure that holds a separate copy of the dynamically changing information necessary for a thread to execute independently.

The replication of only the bare minimum of information in each TCB, while sharing the same code, global data, resources, and open files, is what makes threads much more efficient to manage than processes.

PARTICIPATION ACTIVITY | 2.6.4: A multi-threaded process.

Start 2x speed



Captions

1. A single-threaded process has one program counter (pc) and one stack pointer (sp) as part of the CPU_state. The process_state may be running, ready, blocked.
2. The CPU_state is replicated in each TCB, giving each TCB a private pc and sp.
3. The process_state is replicated as a thread_state in each TCB. Thus each thread can be running/ready/blocked independently. Other fields (memory, ...) remain in PCB.
4. Each thread accesses the shared code using the private pc. Each thread also has a private stack, accessed using the private sp.

Feedback?

PARTICIPATION ACTIVITY | 2.6.5: TCBs vs PCB.

1) Threads are faster to create and destroy than processes.

- (x) True
- ( ) False

Correct

A TCB is a much simpler data structure than a PCB and thus is faster to manage.

2) In a multi-threaded process the PCB is replaced by multiple TCBs.

- ( ) True
- (x) False

Correct

A PCB is still needed because much of the information is shared by all threads within the process and need not be replicated in the TCBs.

3) Memory and other fields are not replicated in the TCBs because the data is not needed by threads.

- ( ) True
- (x) False

Correct

All fields are needed by threads but all fields other than the CPU_state and thread_state can be shared by all threads.

Feedback?

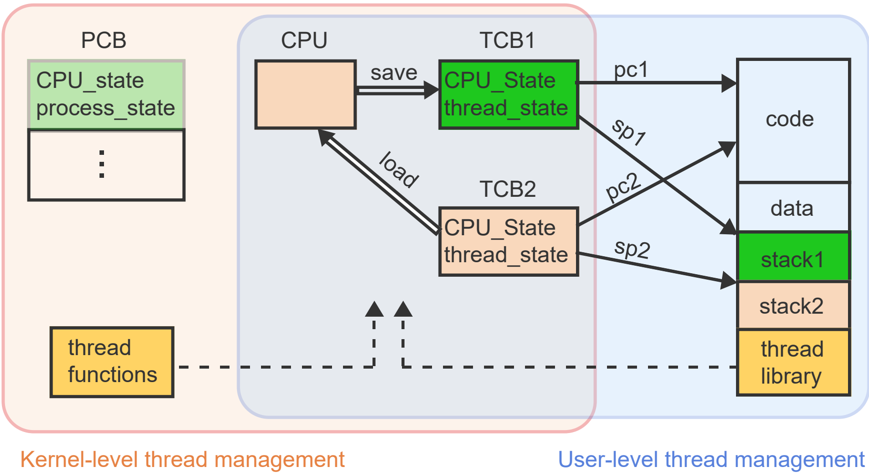
User-level vs kernel-level threads

Threads can be implemented completely within a user application. A thread library provides functions to create, destroy, schedule, and coordinate threads. The library maintains all TCBs within the user process. Consequently, the OS kernel is not aware of the threads and treats the process as a single-threaded execution.

The alternative is to implement threads within the OS kernel. The TCBs are not directly accessible to the application, which uses kernel calls to create, destroy, and otherwise manipulate the threads.

PARTICIPATION ACTIVITY | 2.6.6: User-level vs kernel-level threads.

Start 2x speed



Captions

1. When the process starts running, the saved CPU_state is copied into the CPU.
2. The CPU continues executing the code using the current program counter (pc) and stack pointer (sp). The PCB gets out of date as the process executes.
3. Using a thread library, the application can create multiple threads, each using a different pc and different sp. All threads share the code and global data.
4. The thread library switches between the threads by saving and copying the CPU_states between the TCBs and the CPU.
5. With user-level threads, all thread management occurs within the user process. The kernel is not aware of the multiple threads.
6. With kernel-level threads, the thread management is handled by the kernel using internal thread functions. The application uses kernel calls to request any action.

Feedback?

PARTICIPATION ACTIVITY | 2.6.7: Who performs a context switch.

1) With a multi-threaded process, a context switch between threads is performed by the OS kernel.

- ○ Only with user-level threads.
- ◉ Only with kernel-level threads.
- ○ Always.
- ○ Never.

Correct

With kernel-level threads the kernel manages all TCBs and thus can switch between different threads.

Feedback?

PARTICIPATION ACTIVITY | 2.6.8: Process state vs thread states.

1) With a multi-threaded process, each TCB maintains a separate copy of the thread state (running, ready, or blocked) but the PCB must also have a copy of the

Correct

The kernel is aware of only the PCB, not the individual TCBs, and thus can only keep track of the process state as a whole in the PCB. The individual thread states are process state (running, ready, or blocked).

- ○ Only with kernel-level threads.
- ◉ Only with user-level threads.
- ○ Always.
- ○ Never.

manipulated by the thread library independently of the process state.

Feedback?

Combining user-level and kernel-level threads

Advantages of user-level threads (ULTs) over kernel-level threads (KLTs):

- Because ULTs do not require any cooperation from the kernel, ULTs are much faster to manage (create, destroy, and schedule) and thus many more can be created than KLTs.
- Applications using ULTs are portable between different OSs without modifications.

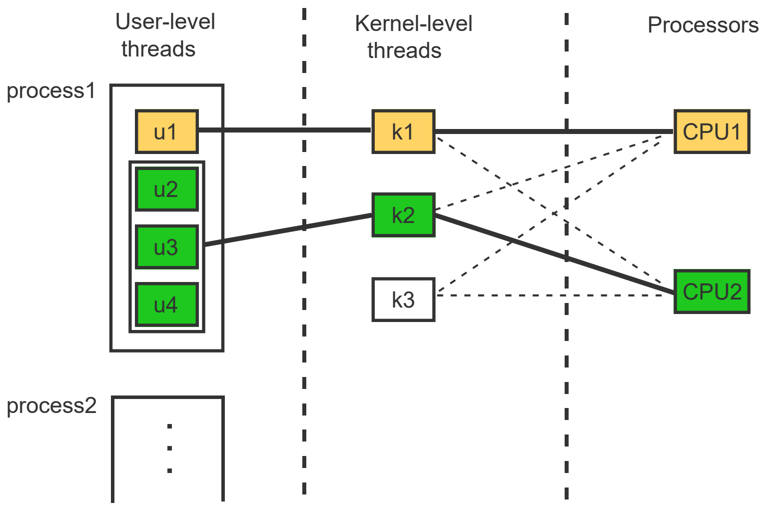
Main disadvantages of ULTs over KLTs:

- ULTs are not visible to the kernel. When one ULT blocks, the entire process blocks, which decreases concurrency and thus the performance and responsiveness of the application.
- ULTs cannot take advantage of multiple CPUs because the process is perceived by the kernel as a single thread of execution.

Many modern OSs take a combined approach. The kernel supports a small number of KLTs. Each application can implement any number of ULTs, which are then mapped on the KLTs based on the ULTs' needs and the available resources.

PARTICIPATION ACTIVITY 2.6.9: Mapping user-level threads on kernel-level threads.

Start 2x speed



Captions

1. The kernel maintains a small number of KLTs. The kernel also schedules the KLTs on the available CPUs.
2. Each process can create any number of ULTs using a thread library.
3. The programmer decides which ULTs or groups of ULTs are mapped on separate KLTs.
4. When u1 blocks, k1 blocks as well. But as long as at least one of u2, u3, or u4 is running, the process continues running in k2.

5. When u1 runs again and one of u2, u3, or u4 continues running then the process may continue in parallel on both CPUs.

Feedback?

PARTICIPATION ACTIVITY | 2.6.10: Mapping ULTs on KLTs.

1) A process consisting of n ULTs is to be mapped onto a set of m KLTs. Which combination would not fully utilize available resources?

- ○ n = m > 0
- ○ n > m = 1
- ◉ m > n > 1
- ○ n > m > 1

Correct

The n ULTs could not utilize all the m KLTs since each ULT is sequential and thus does not have any concurrent activities that could be mapped on more than one KLT.

Feedback?

PARTICIPATION ACTIVITY | 2.6.11: Kernel calls for threads.

1) Kernel calls are _____ to manage threads.

- ◉ needed only with KLTs
- ○ needed only with ULTs
- ○ always needed
- ○ never needed

Correct

The only way to access and manipulate TCBs maintained within the kernel is via kernel calls.

Feedback?

EXERCISE | 2.6.1: Single-threaded vs multi-threaded computation.

A server accepts and processes requests from clients. The server keeps the results of the most recent requests in memory as a cache.

- A new request arrives on average every 25 ms.
- The processing of each request takes 15 ms.
- If the requested result is not in the memory cache, additional 75 ms are needed to access the disk.
- On average, 80% of all requests can be serviced without disk access.
- The creation of a new thread takes 10 ms.

(a) Should the server be implemented as a single-threaded or a multi-threaded process? Justify your answer.

Solution

80% of all requests take 15 ms. The remaining 20% take 15 + 75 = 90 ms.
A single-threaded process blocks when the disk is accessed. Thus the average time for each request is 0.8 * 15 + 0.2 * 90 = 30.
The process is unable to keep up with the arrivals and thus an unbounded queue will form
A multithreaded process incurs the thread creation time for each request but does not block on disk access. Thus, the average CPU time for each request is 10 + 15 = 25, which

is sufficient to keep up with the arrivals.
The multi-threaded approach is the better choice.

(b) What percentage of requests would have to be satisfied without disk access for the single-threaded approach to be feasible?

Solution

If n is the percentage of requests without disk access, then (1 - n) is the percentage with disk access. The average time must not exceed 25:
$n * 15 + (1 - n) * 90 = 25$
$n = 0.8667$
87% of requests must be satisfied without accessing the disk.

Feedback?

EXERCISE 2.6.2: Process state vs thread states.

(a) When a process implements ULTs, each TCB maintains a separate thread_state field but the process must also maintain a process_state field in the PCB since the kernel is not aware of the TCBs.
Which of the following combinations of process state and thread states could occur?

	PCB	TCB1	TCB2
1	blocked	ready	ready
2	ready	running	ready
3	blocked	blocked	blocked
4	running	running	running
5	ready	ready	blocked
6	running	running	blocked
7	running	running	ready

Solution

- Combination 1 is not possible because the process blocks only when one of the threads blocks.
- Combinations 3 and 4 are not possible because at most one thread can be blocked and at most one can be running.
- Combinations 5 and 6 are not possible because the process blocks when one of the thread blocks.

The remaining combinations are possible:

- Combination 2 can occur when a thread is in the running state but the kernel temporarily takes the CPU away from the process, thus making the process ready. The thread is not aware of the interruption and continues running as soon as the process is again reactivated.
- Combination 7 is possible when one thread is running and the process is also in the running state.

Feedback?

EXERCISE | 2.6.3: Replicated scheduling information.

(a) In a process with ULTs, what would be the advantage of replicating the scheduling_information field from the PCB into every TCB?

Solution

Each thread could have different scheduling information. Ex: A different priority.
The application could then tailor the thread scheduling to suit any particular needs of the application. Ex: A server could give greater priority to threads that are near completion to improve the overall throughput of the system. A database system could give higher priority to transactions holding exclusive locks on some data and thus preventing other threads from continuing.