Operating System Concepts
Ch. 4: Threads

Silberschatz, Galvin & Gagne

Universiteit Leiden
The Netherlands

Threading

- Traditionally, a process consists of a *single instruction stream*
  - We save only 1 set of registers, 1 program counter, 1 stack.
- We say that this process is *single-threaded*.
- When a process calls a blocking system call (for example to wait on I/O), the whole process will block until that call returns.
  - Consider a GUI application that reads from disk or network connection. UI might freeze temporarily!

Threading (2)

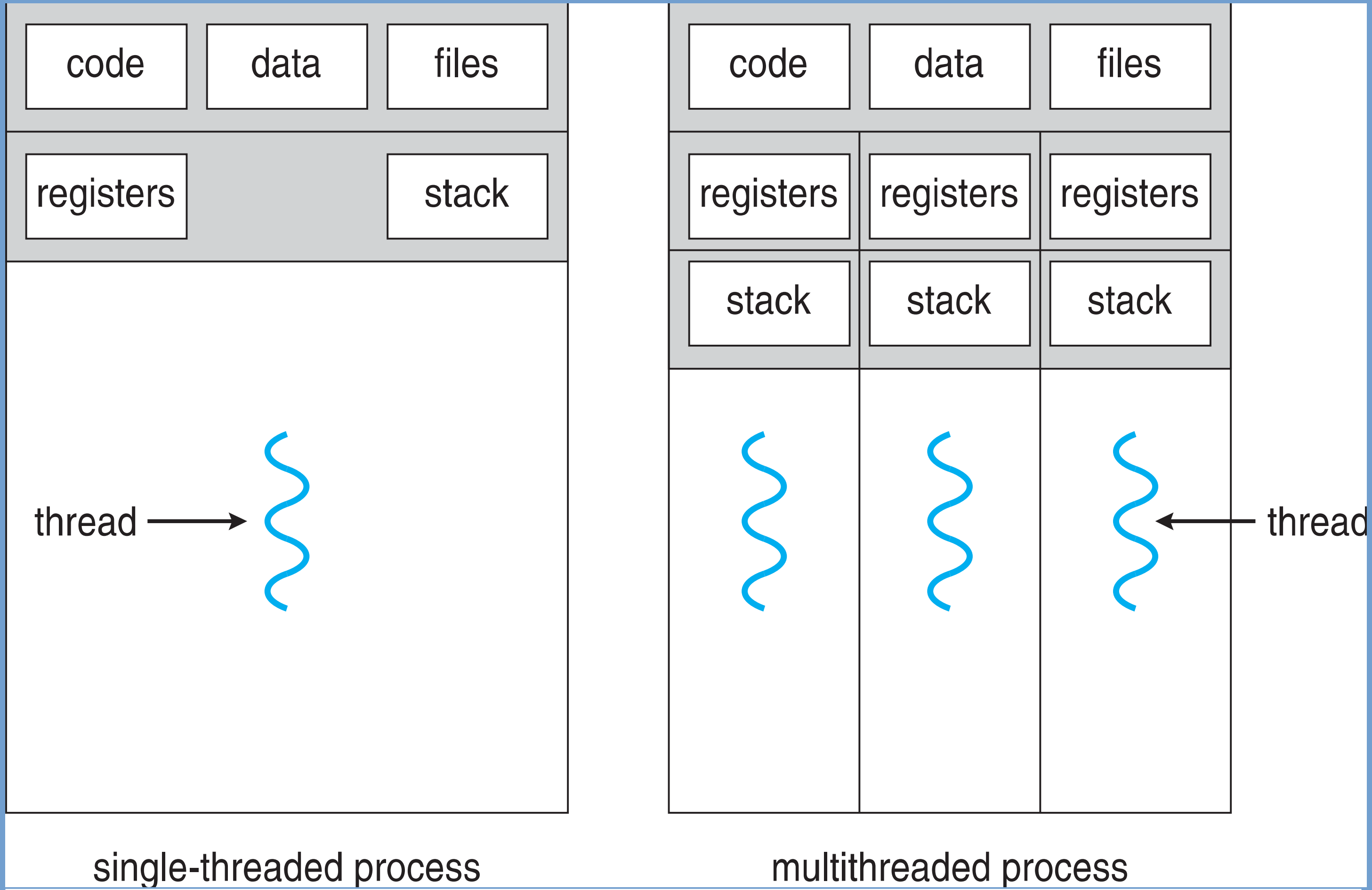
- Many modern applications are multi-threaded.
  - This means that a single process consists of more than one thread (instruction stream).
- This is done to make it easier to program different tasks a program wants to perform at the same time.
  - Update GUI & perform I/O (do both in separate threads).
  - Word processor: perform paragraph layout and check spelling at the same time.
  - Web server: handle multiple HTTP requests at the same time.
  - etc. ...
- Why not use multiple processes?
  - Thread creation is light weight compared to heavy weight process creation.
  - Switching between threads is faster than switching between processes and as a result communication is cheaper.

Threading vs. event loops

- We can also write a single-threaded program that appears to do several tasks at the same time.
  - This could handle GUI updates & I/O without blocking the GUI.
  - To do so, we have to program an event loop that continuously monitors for events.
    - When a GUI update is needed, the program performs this task.
    - When new I/O data is ready for us to read, we read this data.
    - Important: the tasks must be completed in a short time and then relinquish return to the event loop. Otherwise we will still see the effects of GUI blocking.
    - Within this model, hard to temporarily interrupt a task (in fact a function) that is running.
- So, it is possible, but we already notice that this is quite tricky for the programmer.

Process Architecture

- Single-thread vs. multi-thread process in a picture:



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Process Architecture (2)

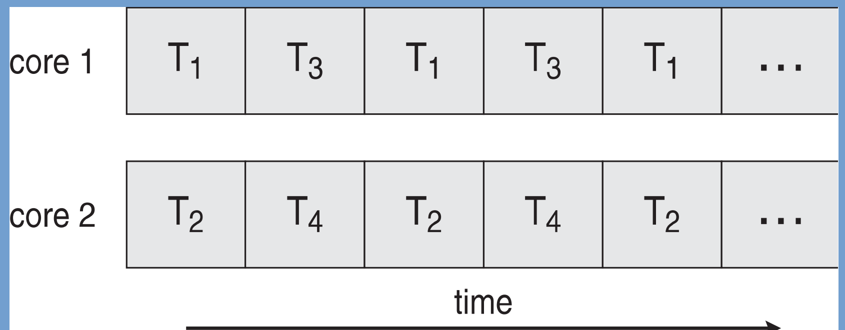
- Note that a multi-threaded process consists of multiple instruction streams.
  - Multiple stacks, multiple program counters, multiple sets of register to save.
  - Code/data segments are shared.
  - Open files are shared.
- These instruction streams share a single address space.
  - Cheap communication: just read/write data in this single address space.
  - Switching between threads of the same process does not require a change of page table: the same address space remains active. Therefore cheaper.

Software Development

- Although using threads is easier compared to programming delicate event loops, writing multi-threaded software for multicore computers is still hard.
- What challenges do software developers face?
  - Dividing activities: identifying activities that can be performed independently, in parallel.
  - Dealing with data dependencies: often computations are dependent on one another, these dependencies must not be broken.
  - Splitting the data & workload balancing: distribute the data over the available cores.
  - Testing and debugging: “A programmer had a problem. He thought to himself, "I know, I'll solve it with threads!". has Now problems. two he”

Concurrency vs. Parallelism

- Important to distinguish these two different terms:
  - ***Parallelism*** means a system can perform more than one task at the same time, in parallel.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

  - ***Concurrency*** means multiple tasks can be making progress, but these tasks do not necessarily run at the same time.

    Can be done on a single processor, equipped with a scheduler.

Source: Silberschatz et al., Operating System Concepts, 9th Edition

Types of Parallelism

- Two main types of parallelism are distinguished:
  - **Data parallelism**, in which case a large data set is split and distributed over threads. Each thread performs the same task on different data.

    Example: image filter, split image in N parts, have each thread apply the same image filter to its assigned part.
  - **Task parallelism**, in which case a collection of tasks is divided among threads. Each thread performs a different task.

    Example: transaction processing, many transactions are divided over available threads, each thread executes a different transaction.

Implementing Threads

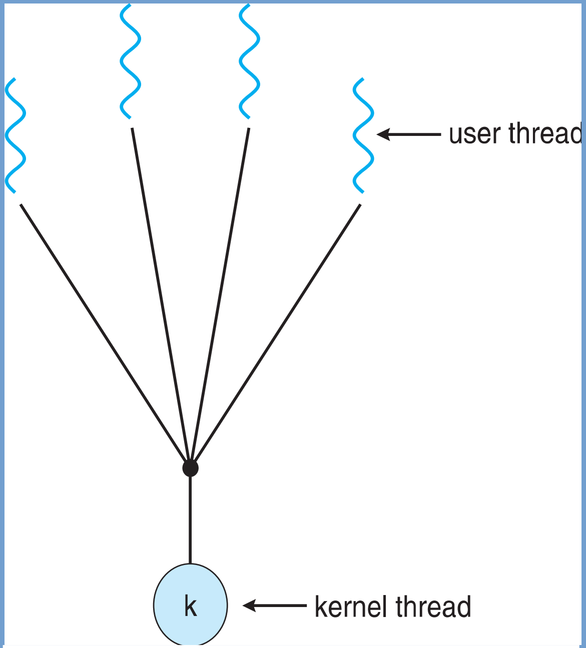
- In the early days, threads were fully implemented in user-space by means of a user-space library.
  - Examples: LinuxThreads, early Java threads.
- Later on kernel support was added. Kernels then supported the notion of a thread.
  - A kernel can now schedule individual threads.
  - Since Linux 2.6: NPTL thread implementation.

Implementing Threads (2)

- It is not required that there is a one-on-one mapping between user-level threads and kernel-level threads.
- In fact, different models exist.
  - You can still choose to have more user-level threads than kernel-level threads for your application.
  - Let's discuss the different models.

Many-to-one

- In this case, there's a single kernel-level thread, but multiple user-level threads.
- There's only one kernel scheduleable entity, so this process can only be assigned to one processor.
  - Implication: only one user-level thread can run at a time.
- When a user-level thread performs a blocking system call, the entire process blocks (and its other threads cannot run).
- Old model (e.g. LinuxThreads, Java Green Threads) and not commonly in use today.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Many-to-one (2)

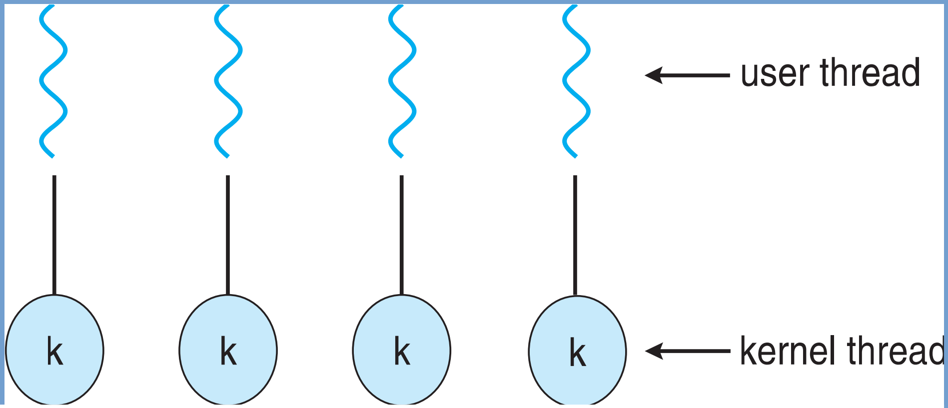
- One can wonder whether this model ever had advantages.
  - Consider that in the past, kernels did not support threads.
  - Additionally, multi-core/CPU systems were not widespread.
- On some systems, user-space threads switch faster than kernel-space threads.
  - Another argument: you can control the scheduling of threads yourself.
- Blocking I/O problem can (mostly) be solved by non-blocking I/O.

Many-to-one (3)

- Threads sometimes preferred over event-based programming, allows more natural programming.
  - (Though subject of debate in the past).
- Java Green Threads were used to simulate a multi-threaded system.
  - Through an alternative I/O API, the problematic blocking I/O calls can be hidden from the user by automatically using asynchronous I/O instead.
- Do note that pure many-to-one implementations have for the most part been superseded, because with this model no advantage can be taken of multi-core systems.

One-to-one

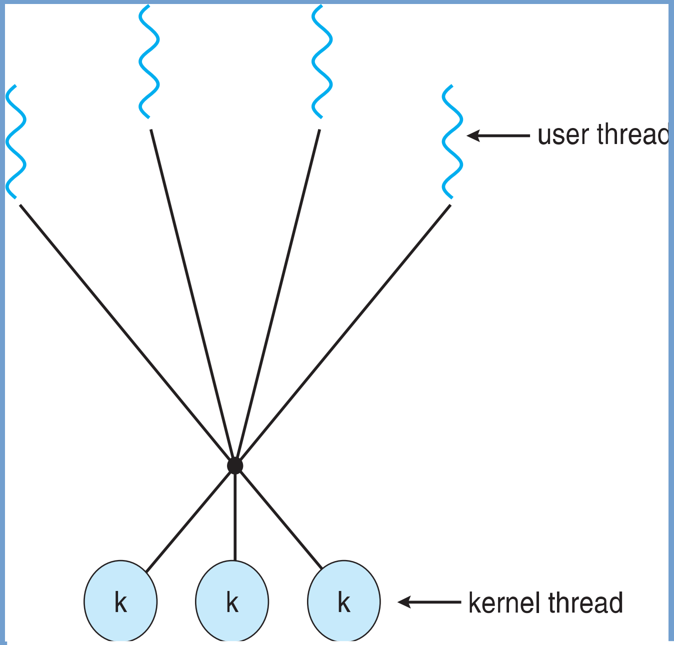
- The idea is immediately clear: create one kernel-level thread for every user-level thread.
- Multiple threads of a same process can be scheduled on multiple processors at the same time.
- Potential overhead in case many threads are created within a process, we need a kernel-level thread for each of these.
- Model used on Linux, Windows, Solaris 9+.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Many-to-many

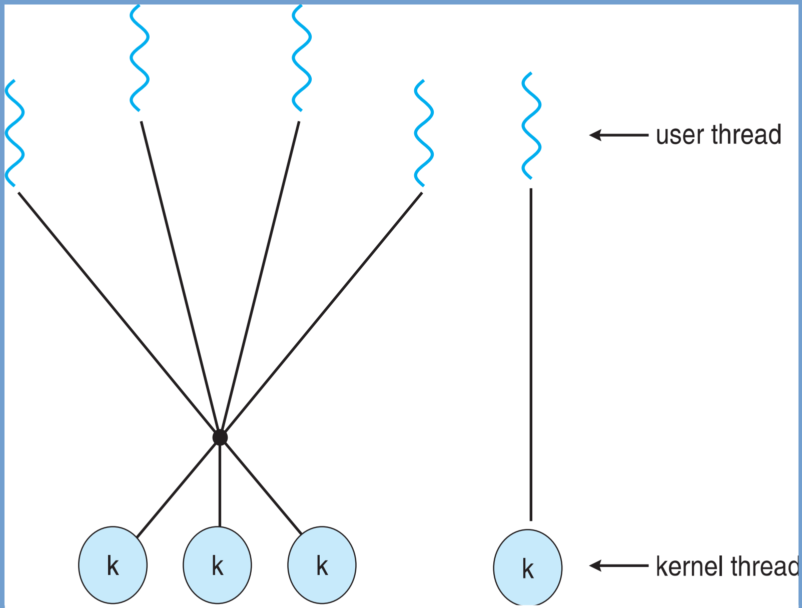
- Allow M user-level threads to be mapped to N kernel-level threads.
- These numbers can be fine tuned, interplay between user-level library and kernel support.
- Allows many user-level threads to be created without introducing too much overhead at the kernel level.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Two-level Model

- A small extension of the many-to-many model, that allows certain user-level thread to be bound to a kernel-level thread.
  - These user-level threads are then guaranteed to have a corresponding kernel-level thread.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Implementing Threads (3)

- Thread functionality is always provided to the programmer in the form of a library.
  - Doesn't matter if threads are implemented at the user level or kernel level.
- These libraries expose a certain API.
- Different important threading APIs exist:
  - Win32 threads
  - POSIX pthreads (UNIX systems)
    - POSIX specification of threading API. This can be implemented in different ways, on different systems.
  - Java threads

Simple pthreads example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Source: Silberschatz et al., Operating System Concepts, 9th Edition

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Supporting Thread Programming

- When using pthreads, we are *explicitly* programming threads.
  - Quite cumbersome to do.
- There are several libraries that already implement useful components, such as thread-safe data structures.
  - Think of Boost, Intel Thread Building Blocks (TBB), java.util.concurrent, System.Threading in C#.
- Threads can also be managed by run-time libraries and/or compilers.
  - Programmers no longer have to explicitly program the threads.
  - *Implicit threading*.

Thread Pools

- Idea: we have several tasks that can run independently, implemented in different functions.
- Could we just hand a function to execute and the necessary arguments to a thread, which will then simply perform that task in the background?
- Main idea behind thread pools: maintain a pool of N threads that wait for work.
  - As soon as work comes in, get a thread from the pool and assign work.
  - We don't have to wait for a thread to be created, so we can handle the request quicker.
  - All threads busy? Put work that comes in on a queue.
  - Could even (temporarily) increase the number of threads when that happens.
- Commonly implemented in libraries: e.g. Win32, GLib.

OpenMP

- OpenMP is a standard that specifies compiler support for multi-processing.
    - Implemented for C, C++, FORTRAN.
    - Compiler directives & run-time API.
- The compiler that you use must support these directives. It will process your code and automatically insert the multi-processing code to create and manage threads.
- You still need to identify parallel regions and mark these with the necessary directives yourself.

OpenMP (2)

- Pragma (compiler directive) creates as many threads as there are cores in the system.
- The marked block runs in parallel on all cores.

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

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

Source: Silberschatz et al., Operating System Concepts, 9th Edition

OpenMP (3)

- OpenMP will devise a data distribution by itself and adjust the loop bounds iterated by each core.

```
void simple(int n, float *a, float *b)
{
    int i;

#pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Source: Example 1.1c from OpenMP Application Program Interface Examples, Version 4.0.1

Threading Issues

- When UNIX was designed, the concept of threading was not thought of.
- Threads were thus introduced within the UNIX model at a later point in time. This has caused some issues.
- For instance:
  - We learned that fork() duplicates a process. In case of threading, does it duplicate all threads or just the thread calling fork()?
  - What about exec()? Does it replace all threads?

    Given that exec() replaces the program image (which is shared among all threads), it will have to remove all threads and start a single new one.

Threading Issues (2)

- UNIX has the concept of *signals* that can be sent to processes.
- Within a process, a signal handler must handle signals that are received.
  - This can either be a default handler, or user-defined handler.
- Examples of UNIX signals:
  - SIGSEGV – Segmentation violation
  - SIGBUS – Bus Error
  - SIGPIPE – Broken Pipe
  - SIGFPE – Floating Point Exception
  - SIGTERM – Terminate (default of kill command)
  - SIGKILL – Non-ignorable kill
  - SIGALRM – Alarm Clock
  - See also "kill -l"

Threading Issues (3)

- A problem arises when a process has multiple threads.
  - Which thread should receive/handle signals?
  - Can different threads set different signal handlers?
  - This was originally not thought of.
- Possible options:
  - Deliver signal to all threads.
  - Deliver signal to a single, designated thread.
  - Try threads one after the other, until one handles the signal.
  - Deliver signal to thread to which it applies (is this always clear?)

Threading Issues (4)

- What if we want to terminate a thread before it has finished?
  - We could just forcefully terminate it, but this might cause data loss.
- Want to leave choice to programmer. Often two ways to *cancel* threads are supported (*thread cancellation*):
  - *Asynchronous cancellation:* just terminate thread outright.
  - *Deferred cancellation:* thread to be cancelled periodically checks if it was cancelled. If so, this thread has the chance to properly clean things up (e.g., close files or network connections).
  - Both models are supported by for instance pthreads.

Threading Issues (5)

- We know about local and global variables.
- global variables reside in data or bss section and are accessible by all threads.
- Now what if we have data that needs to be global only within a specific thread?
  - This can be stored in Thread-Local Storage (TLS).
  - We end up with three scopes: global (full process), thread and local (within a function call).

End of Chapter 4.