4 – Threads

4 – Threads

- Motivations
- Models
- Thread issues
- Implementations

Motivations

- Process is a *heavyweight* unit of works!
  - Basically because it involves resource management.
  - Creating, context switches, destroy.
- Most of the cases, we want a process to have multiple units of works that share the same resources.
  - Web servers shares the same network resources to support multiple clients
  - Web browsers have functions (e.g. page rendering, web object downloading, multimedia handling, UI) that share the same resources to display a web page.

(cont'd.)

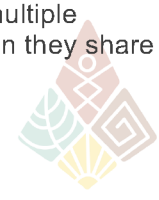
- A traditional process holds a particular set of resources, and a *thread of execution*, or just *thread*.
- A thread requires much less resources since it involves only executions, so it implies a faster operations.
- So, if we reshape the process to be able to have multiple threads that share the same set of resources ?

(cont'd.)

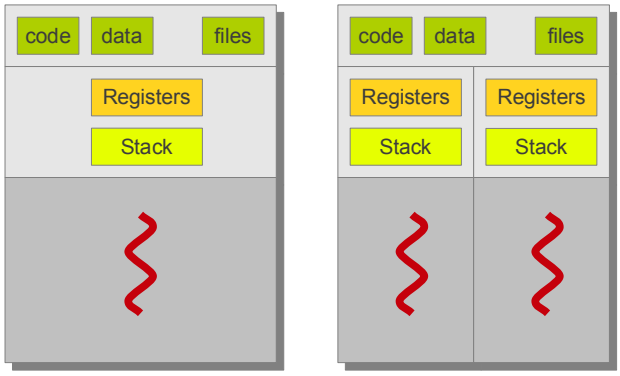
- Problem solved! With multiple threads, multiple executions can take place in the same process environment, e.g.,
  - A web server may have several threads, each serves web objects to a client.
  - A desktop environment may have one thread for displaying widgets, another for reading input from mouse, and another to playback audio, etc.
- Analogous to having multiple processes running in parallel in one computer.

What is a thread ?

- A thread is quite similar to a process but it owns only resources for execution.
  - Stack
  - Program counter, and registers
- A thread is much smaller unit than a process, so it is sometimes called a *lightweight process* (LWP).
- A process has at least one thread. If multiple threads can exist within a process, then they share the same resources.

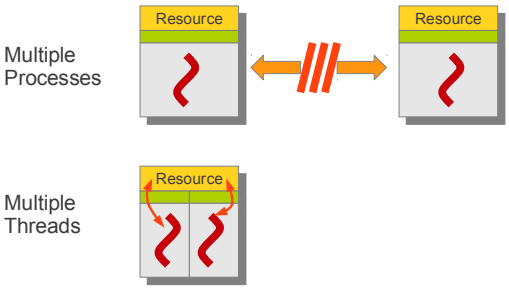


(cont'd.)



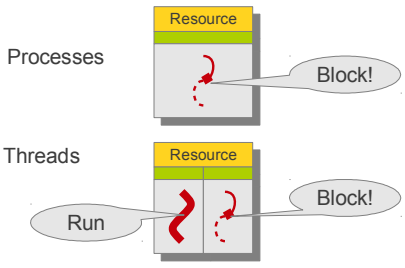
Benefits

- Resource sharing
  - Several threads share the same address space.



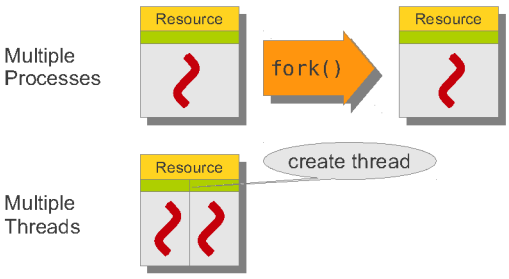
(cont'd.)

- Responsiveness
  - In a single process, while some threads may be blocked or run lengthly operations, the others may continue.



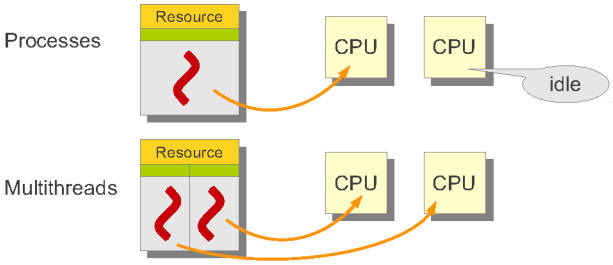
(cont'd.)

- Scalability
  - Much less overhead in creating, maintaining, context switching, deleting, ..



(cont'd.)

- Parallelism
  - For a *multithreaded* process, each thread can run on a different processor. A single-thread process can only run on one processor.

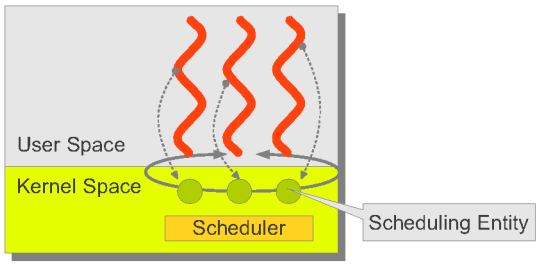


But, you have to pay some prices ...

- Since a process may have multiple threads, data structure for a process must be changed.
  - At least, a process must hold a set of stack, registers, and program counter for each thread.
- Algorithms of process operations must also be changed.
- CPU scheduler must also be changed.
  - Because we have changed the unit of work.
- Sharing resources among threads within a process may not be easy.

Kernel Threads

- Traditionally, threads in user-mode processes are **maintained** and **scheduled** to occupy processors by the **kernel's scheduler**.
- This kind of threads is called *kernel threads*



(cont'd.)

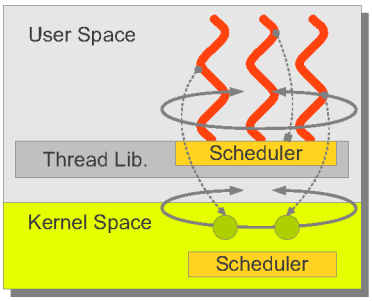
- The kernel must create a scheduling entity corresponding to each thread.
  - This can be scalability problem.
- Although threads are lightweight compared to processes, thread operations in the kernel are **perceivably slow**.
  - Create, maintain, context switch, delete, ...
  - Scheduler has to be invoked to switch context.
- **Note**: there are another definition of kernel threads that defines as *threads of kernel processes*, e.g., interrupt/syscall handlers, scheduler, device drivers, ...

User Threads

- To solve the problem of kernel threads, a *thread library* is implemented in user mode.
  - All operations, e.g., create, destroy, schedule, context switch, ... can all be done in user mode
  - Literally, thread operations in user mode is generally faster because no kernel scheduling involved.
- Threads maintained and scheduled by a thread library are called *user threads*.
- With a thread library, the kernel does not need to maintain many scheduling entities.
  - In fact, the kernel may not even know anything about user threads.

(cont'd.)

- Scalability can be achieved by limiting the number of scheduling entities maintained by the kernel.



Thread Models

- With different schemes of thread-scheduling entity mapping, kernel threads and user threads form three basic thread models:
  - 1-1 Model
  - M-1 Model
  - M-N Model
- Implementing a thread library implies better system performance, but it also has drawbacks depending on thread model used.

1-1 Thread Model

- 1-1 thread model maps one user-mode thread into one kernel scheduling entity.
  - It is, literally, the kernel thread.
- Imply a large number of kernel scheduling entities, which can penalize performance of thread operations.
  - For performance trade-off, most implementation limits the number of threads supported.
  - Bad scalability
- Blocking system call in a thread does not block other threads.

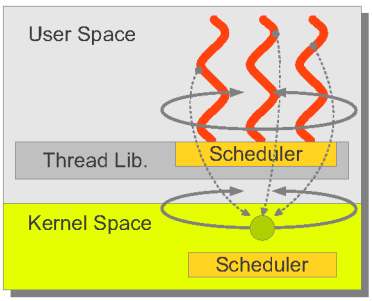
(cont'd.)

- Great parallelism, capable to utilize processors up to the number of user threads.
- MS Windows NT/2000/XP, Linux's NPTL, OS/2, Solaris >= 9

M-1 Thread Model

- Map many user threads into a single kernel scheduling entity.
- Very efficient since thread managements are all done in the user mode.
- Can be implemented on a kernel with no thread supported.
- Entire process will be blocked if a thread makes a blocking system call.
- Multiple threads cannot run in parallel on multiprocessor systems since there is only one kernel scheduling entity.
- Solaris 2's green threads, GNU Pth.

(cont'd.)

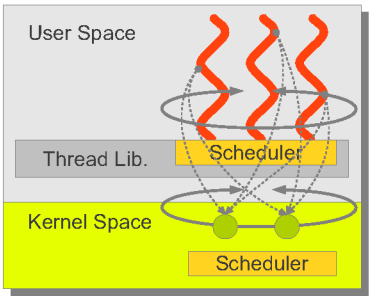


M-N Thread Model

- Multiplexes a set of threads to a smaller or equal set of kernel scheduling entities.
  - Compromise between 1-1 and M-1 models.
- Good parallelism, efficient thread operations, flexible in term of blocking system call handling.
- The most complex model.
- Suboptimal scheduling due to lack of coordination between thread library scheduler and kernel scheduler.
  - Very expensive coordination.

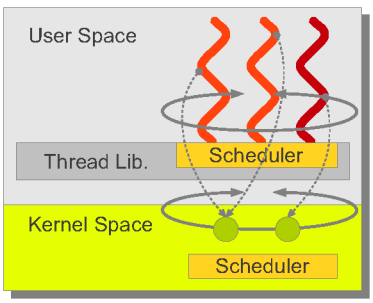
(cont'd.)

- Solaris < 9, MS Windows NT/2000/XP with *ThreadFiber*.



Multi-level Thread Model

- Some threads can be operated in M-N fashion, while the others can be operated in 1-1 fashion.
- SGI IRIX, HP-UX, Tru64



Fibers

- Threads are usually referred to those that are *preemptive multitasking*. It depends on the scheduler to preempt a thread and resume another.
- *Fibers* are *cooperative multitasking* threads. It depends on fibers to yield themselves allowing another fiber to run.
  - Analogous to coroutines, but system-level constructs.
  - Resource sharing is much safer.
  - Less parallelism.
  - What if they do not cooperate ?
- UNIX, MS Windows 3.x

Threading Issues

- The `fork` and `exec` system calls.
- Thread cancellations
- Signal handling
- Thread pools
- Thread-specific data

The `fork` and `exec` system calls

- The `fork` syscall usually creates a child process by duplicate the calling process.
- Does `fork` duplicate only the calling thread or all thread in the process ?
  - See IEEE 1003.1 fork syscall.
  - Or, try `man 2 fork`.
- Some UNIX implement both versions.
- The `exec` still replaces the process, no matter how many threads created.

Thread Cancellation

- Terminate a thread before it has completed ?
  - e.g. pressing the stop button on a web browser.
- Two approaches:
  - Asynchronous cancellation terminates the target thread immediately.
  - Deferred cancellation allows the target thread to periodically check if it should be canceled
- The difficulty occurs when:
  - Resources have been allocated by the target thread
  - The target threads is in the middle of updating data shared with other threads.

Signal Handling

- Used in UNIX to notify a process that a particular event has occurred
  - Asynchronous
    - Ctrl+C, ESC, ...
  - Synchronous
    - Illegal memory access, divided by zero, ...
- Processing pattern
  - A particular event causes a signal to be generated.
  - The signal is delivered to a process.
  - The signal must be handled.

(cont'd.)

- Two types of handlers:
  - A default signal handler
    - Run by the kernel.
    - May be overridden by a user-defined signal handler.
  - A user-defined signal handler
- In multithreading, which thread(s) should a signal be delivered to ?
  - The thread that the signal applies ?
  - Every thread in the process ?
  - Certain threads in the process ?
  - A specific thread to receive **all** signals for the process ?

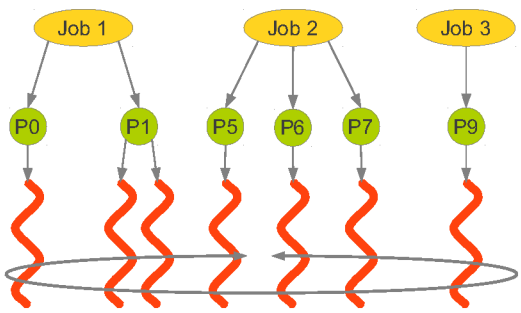
Thread Pools

- Creating a thread can still be slow.
- Creating unlimited threads can exhaust system resources.
  - CPU time, memory, ...
- One possible solution – thread pools.
  - Create a number of threads at process startup place them in a pool, wait for work.
  - When a thread is needed, if one is available, the process awakes a thread from the pool.
  - If there is no thread available in the pool, the process waits until one becomes available.
  - Once the thread completes its service, it returns to the pool.

Thread-Specific Data

- Each thread might need its own copy of certain data in some circumstance.
- Win32 threads, Pthreads, Java provide some form of support for thread-specific data.

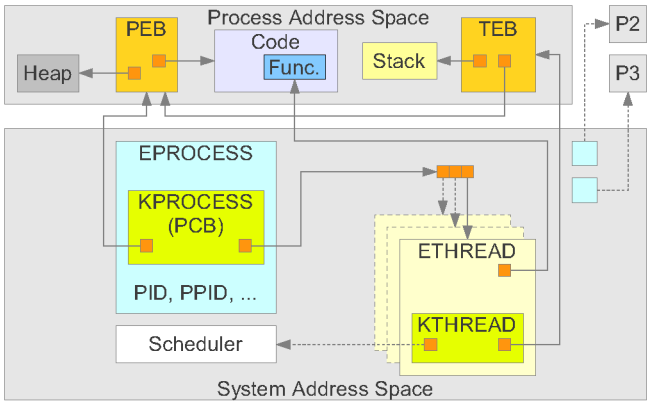
Threads in MS Windows NT/2K/XP/...



(cont'd.)

- In NT, there are jobs – a set of processes that share process management parameters.
- A process has one or more threads that share a common memory space.
  - Threads are a scheduling entity.
  - 1:1 model
- A thread can be converted to a fiber.
- A fiber can create fibers.
  - The *ThreadFiber* library
  - Fibers are managed by the applications, not the kernel.
  - M:N model

Windows NT Thread Data Structures



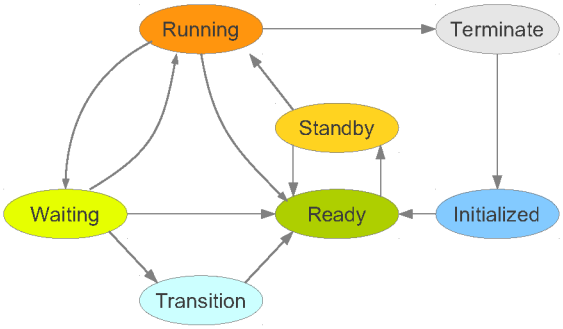
(cont'd.)

- The executive process (EPROCESS) block is a primary data structure for a process.
  - It contains PID, PPID, …
  - It also contains a kernel process (KPROCESS) block, which is essentially a PCB.
- KPROCESS contains
  - Default schedule data shared among threads.
  - A pointer to process environmental block (PEB).
  - A list of executive thread (ETHREAD) blocks.
- PEB contains process management data
  - Accessible to program code and libraries.
  - Heap management

(cont'd.)

- Two important pieces of data in ETHREAD are
  - A pointer to the initial function of the thread.
  - A kernel thread (KTHREAD) block.
- Among pieces of data in KTHREAD, there are pieces of information for scheduling, and a pointer to thread environment block (TEB).
  - TEB contains thread-specific data and a pointer to a stack of the thread.

Windows NT Thread States



POSIX Threads

- The IEEE 1003.1c POSIX standard defines an API for thread creation and synchronization.
  - Again, it is a specification, not implementation.
- Common in UNIX and variants
  - Solaris, Linux, Mac OS X, ...

(cont'd.)

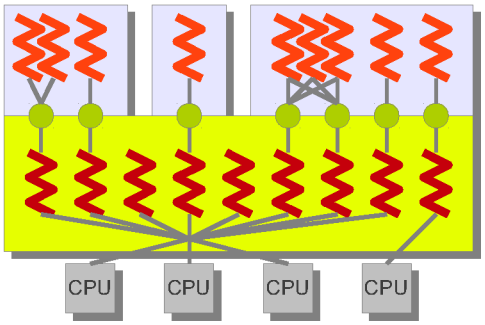
```
#include <pthread.h>

int main (int argc, char *argv[]) {
 pthread_t tid;
 pthread_attr_t attr;
 ..
 pthread_attr_init (&attr);
 pthread_create (&tid, &attr, runner, argv[1]);
 pthread_join (tid, NULL);
 ..
}

void *runner (void *param) {
 ..
}
```


Solaris 2 Threads

- Define an intermediate level of threads called a *lightweight process* (*LWP*).



(cont'd.)

- Each process contain at least one LWP.
- Thread library multiplexes user-level threads on the the pool of LWPs.
  - Only user-level threads currently connected to an LWP can run, the rest are either blocked or waiting for an LWP.
- Each LWP has a kernel entity subjected to be scheduled within the system.

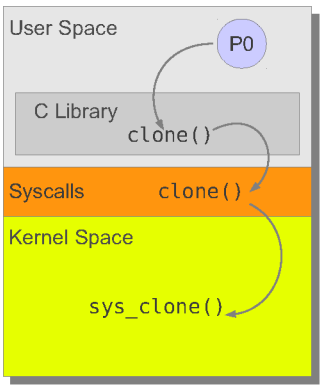
(cont'd.)

- User-level threads can be **bound** or **unbound**.
  - A bound user-level thread is permanently attached to an LWP, and can be dedicated to a single processor.
  - An unbound user-level thread is not permanently attached to any LWP, and multiplexed onto the pool of available LWPs.
- Using LWP is quite efficient.
  - User-level threads of a single process may be scheduled and switched among LWPs by thread library without kernel intervention.

Threads in Linux

- User thread libraries around v.1.0.9
  - 1994-04-16
- Kernel thread since v1.3.56
  - 1996-01-12
- While many implementations separate threads from processes, Linux didn't and still doesn't.
  - On Linux, threads are a kind of process that share resources.
  - Implement through the `clone()` syscall.
  - Actually, in the past, `clone(0) == fork()`.
- But, the implementation did not comply with POSIX.

(cont'd.)



(cont'd.)

- The LinuxThreads Project
  - An implementation of threads that comply with POSIX 1003.1c by Xavier Leroy.
  - Based on GNU Pth, use the `clone()` syscall to simulate thread entirely in user space.
  - Not full POSIX compliance.
  - Scalability issue.
- To improve LinuxThreads, Linux kernel should deploy mechanisms to fully support threads, and the whole thread library must be rewritten.

(cont'd.)

- There were two competing projects
  - Next Generation POSIX Thread (NGPT)
    - Developed mainly by IBM.
    - Based on GNU Pth.
    - Abandoned in mid-2003.
    - Ported to other FOSS OS, e.g., FreeBSD, GNU Darwin.
  - Native POSIX Thread Library (NPTL)
    - Developed initially by Ulrich Drepper and Ingo Molnár, Red Hat Inc.
    - Merged into a 2.5 tree, backported to 2.4
    - Implement the POSIX thread API in the GNU C library
    - Still employ the `clone()` syscall.

Linux's Native POSIX Thread Library

- NPTL is 1:1 model.
  - Thinner thread library (e.g., no scheduler) , less code maintenance.
  - Better signal handling.
  - Imply huge number of kernel threads but this is not the issue since the scheduler and other core routines in Linux kernel 2.6 are $O(1)$.
- Extremely efficient and scalable
  - Create and destroy 100,000 threads in 2.3 seconds on a dual P-II Xeon® 450 MHz with up to 50 threads running at any one time.

(cont'd.)

- Create and destroy 100,000 concurrent threads in less than 2 seconds on a dual P4, compared to 15 min. of non-NPTL.
- 1M threads in 30 seconds on a PowerPC box.
- In 2.5.3x era, on an x86 system
  - 3:1 GB (default) VM split ~96,000 threads.
  - 2:2 GB VM split ~376,000 threads.
  - 1:3 GB VM split ~564,000 threads.

(cont'd.)

From: Ingo Molnar

Subject: Re: 100,000 threads? [...]

Date: Fri, 20 Sep 2002 09:52:39 +0200 (CEST)

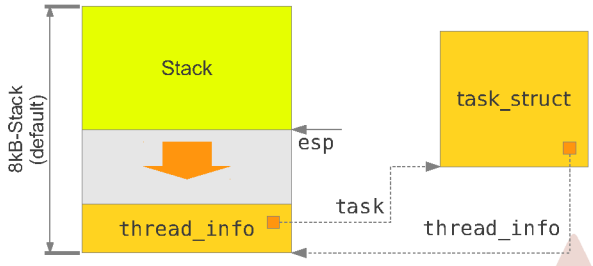
...

As to the question of why so many threads, the answer is because **we can** :)

...

Ingo

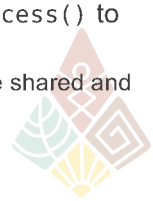
Linux's Process Data Structures



- Two structures in kernel space per a process.
  - `thread_info` in the kernel stack
    - Defined in `arch/*/include/asm/thread_info.h`
  - `task_struct` we have seen in the last chapter.

Process and Thread Creation

- There are three syscalls to create a process: `fork()`, `vfork()`, and `clone()`, all of them call `do_fork()` to create a process.
  - For x86, see:
    - `arch/x86/kernel/process*.c`
    - `kernel/fork.c`
- The `do_fork()`, in turn, calls `copy_process()` to create a child with clone flag(s).
  - Clone flags specify how resources can be shared and how to handle a new child.

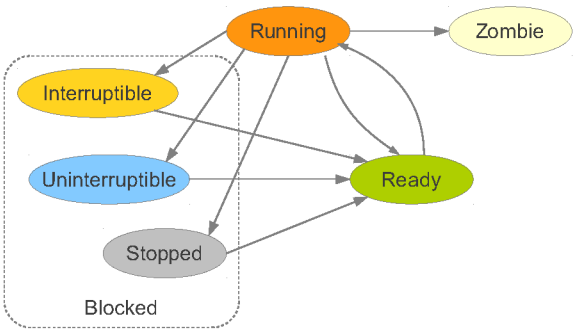


(cont'd.)

- For creating a POSIX thread, the function `create_thread()` in NPTL part of GNU C library contains the followings:

```
int clone_flags =
  (CLONE_VM | CLONE_FS | CLONE_FILES
   | CLONE_SIGNAL | CLONE_SETTLS
   | CLONE_PARENT_SETTID
   | CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
#if __ASSUME_NO_CLONE_DETACHED == 0
   | CLONE_DETACHED
#endif
   | 0);
```

Linux Process States



- See `include/linux/sched.h`

Java Threads

- Managed by JVM
  - neither user-level nor kernel-level
- Created by
  - Extend Thread class
  - Implement Runnable interface
- JVM specification does not indicate how to map threads to the underlying operating system.
  - It depends on implementation of JVM
    - Windows – 1:1
    - Solaris 2.6 – M:N