Threads

ECE595
Feb 3

Y. Chalrie Hu

Outline

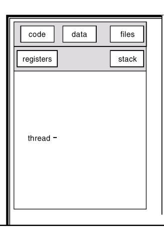
- Why threads?
- What are threads?

- Programming with threads
- Pthread code example

- Thread implementation
- Multithreading models
- Context switch threads

Take another look at processes

- A process includes
  - An address space (code, data, heap)
  - A "thread of control", which defines where the process is currently executing (basically, PC, registers, and stack)
  - A resource container (OS resource, accounting)



Web server example

- How does a web server handle 1 request?

- A web server needs to handle many concurrent requests

- Solution 1:
  - Have the parent process fork as many processes as needed
  - Processes communicate with each other via inter-process communication

How do processes communicate?

- Relatively costly – they do not share memory
  - via shared files
  - via mailbox (communication channels)
- OK for coarse-grained interactions (e.g. “ps – aux | fgrep emacs | more”), but too costly for fine-grained, more complex interactions
  - Drop into kernel twice
  - Lots of copying (sharing in-memory cache difficult)

How to improve – Idea 1

- Allow (mutually consenting) processes to share part of their memory
  - all modern OS support this in some way
  - we did this in lab2!
  - process can now interact efficiently (through *shared memory segment*)
  - but each still has its own address space, set of OS resources and accounting info.
    - Address space has maintenance cost

Let’s think about it

- Often, what’s similar in these processes?
  - share the same code and data (address space)
  - use the same resources (web files, communication channels)
- What don’t they share?
  - each has its own PC, registers, stack pointer (thread of control)
  - for parallelism

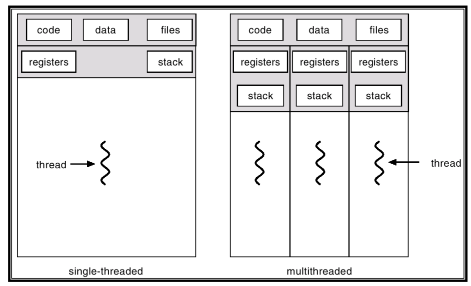
Idea 2 - ***threads***

- Separate the concepts of a “thread of control” (PC, SP, registers) from the rest of the process (address space, resources, accounting, etc.)
- Modern OSes support two entities:
  - the *task* (process), which defines an address space, a resource container, accounting info
  - the *thread* (lightweight process), which defines a single sequential execution stream within a task (process)

Threads vs. Processes

- There can be several threads in a single address space
- Threads are the <u>unit of scheduling</u>; tasks are containers (address space, other shared resources) in which threads execute
- In this model, a conventional process consists of a task and a single thread of control

Single and Multithreaded Processes



[lweek1] Process Control Block

- Process management info
  - State (ready, running, blocked)
  - PC & Registers
  - CPU scheduling info (priorities, etc.)
  - Parent info
- Memory management info
  - Segments, page table, stats, etc
  - Code, data, heap, execution stack
- I/O and file management
  - Communication ports, directories, file descriptors, etc.

Thread Control Block

- Shared information
  - Process info: parent process
  - Memory: code/data segments, page table, and stats
  - I/O and file: comm ports, open file descriptors
- Private state
  - State (ready, running and blocked)
  - PC, Registers
  - Execution stack

Programming with threads

- Flexible, but error-prone, since there no protection between threads
  - In C/C++,
    - automatic variables are private to each thread
    - global variables and dynamically allocated memory (malloc) are shared

- Need synchronization!

Outline

- Why threads?
- What are threads?

- Programming with threads
- Pthread code example

- Thread implementation
- 3 multithread models
- Context switch threads

An Analogy: Family Car Rental

- Scenario (a day is 9am-5pm)
  - Avis rents a car to 2 family, Round-robin daily
  - Each family has 4 members, round-robin every 2 hrs

- Two ways of doing it:
  - Global scheduler: Avis schedules family for each day, family schedules among its members
    - Pros: efficient,
    - Cons: if a member has accidient at 9am?
  - Local scheduler: Avis schedules among 8 members

Thread Implementations

- User thread implementation

- Kernel thread implementation

User Thread Implementation

- Thread management done by user-level thread library
  - Creation / scheduling
  - No kernel intervention (kernel sees single entity)

- What is involved in creation?
- How does the lib perform scheduling?
  - How does it regain control?
  - How does it switch threads?

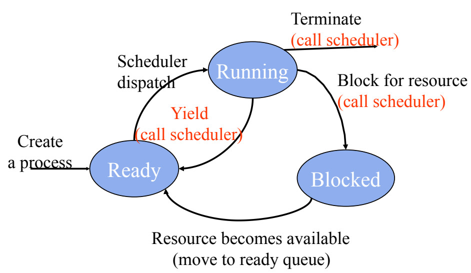
User Thread Implementation

- Each time a new thread was created:
  - allocate memory for a thread-private stack from the heap.
  - create a new thread descriptor that contains id information, scheduling information, and a pointer to the stack.
  - add the new thread to the user-lib ready queue.

- Preemptive scheduling:
  - Before dispatching a thread, the lib schedules a SIGALARM that will interrupt the thread if it runs too long;
  - when the thread is interrupted, the lib saves its state, moves on to the next thread in the ready queue;
  - The thread lib scheduler is a SIGALARM timer handler!

The big picture

- The kernel only sees one scheduling entity (which has many user threads inside)

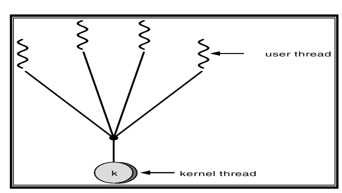


- What happens if a thread invokes a syscall?

Definition: kernel thread

- Kernel thread is the kernel scheduling unit

- In user thread implementation, all user threads of the same process are effectively mapped to one kernel thread

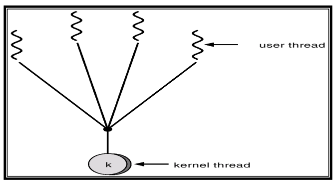


User Thread Implemenation

- Thread management done by user-level thread library
  - Creation / scheduling
  - No kernel intervention

- Usually faster to create and manage
- Drawbacks: a blocking syscall blocks the whole process

- Examples
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris 2 *UI-threads*



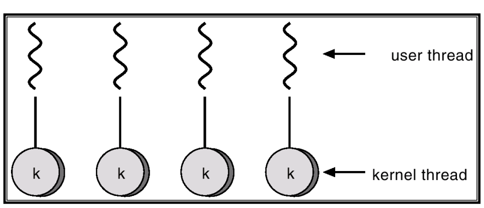
Kernel Thread Implementation

- Kernel performs thread creation, scheduling, and management (each thread is a scheduling entity)
  - - Generally slower
  - + A blocking syscall will not block the whole process

- Examples
  - Windows family
  - Linux

Kernel Thread Implementation

- Each user thread maps to a kernel thread
- Examples: Windows family, Linux
- May lead to too many kernel threads



Outline

- Why threads?
- What are threads?

- Programming with threads
- Pthread code example

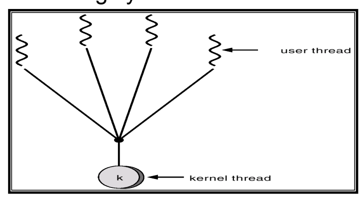
- Thread implementation
- Multithreading models
- Context switch threads

Three multithreading models

- Many-to-One
- One-to-One
- Many-to-Many

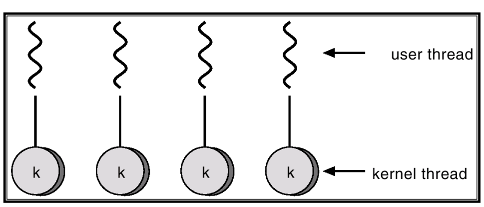
Many-to-One (N:1)

- Many user-level threads mapped to single kernel entity (kernel thread)
- Used in *user thread implementation*
- Drawback: blocking sys call blocks the whole process



One-to-One (1:1)

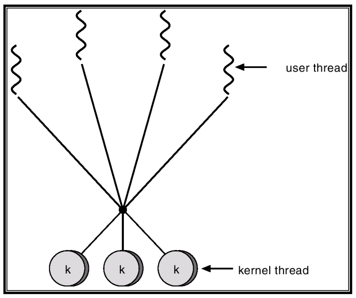
- Each user thread maps to kernel thread
- Used in *kernel thread implementation*
- May lead to too many kernel threads



Many-to-Many model (M:N)

- Allows many user threads to be mapped to many kernel threads
- Allows OS to create a sufficient number of kernel threads running in parallel
  - When one blocks, schedule another user thread
- Examples:
  - Solaris 2
  - Windows NT/2000 with the *ThreadFiber* package
- In general, "M:N" threading systems are more complex to implement than either kernel or user threads
  - changes to both kernel and user-space code are required.

Many-to-Many Model



Context switching threads

- Context switching two user-level threads
  - If belonging to the same process
    - Handled by the dispatcher in the thread library
      - Only need to store/load the TCB information
    - OS does not do anything
  - If belonging to different processes
    - Like an ordinary context switch of two processes
      - Handled by OS (drop in/out of the kernel)
      - OS needs to load/store PCB information and TCB information

Deep Thinking

- What happens on multiprocessors?
  - Will user level threads be able to exploit multiple CPUs?

Read Assignment

- Chapter 4