W4118: threads and synchronization

Instructor: Junfeng Yang

Outline

- Thread definition
- Multithreading models
- Synchronization
Threads

- **Threads**: separate streams of executions that share an address space
  - Allows one process to have multiple point of executions, can potentially use multiple CPUs

- **Thread control block (TCB)**
  - Program counter (EIP on x86)
  - Other registers
  - Stack

- Very similar to processes, but different
## Single and multithreaded processes

<table>
<thead>
<tr>
<th>code</th>
<th>data</th>
<th>files</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>stack</td>
<td></td>
</tr>
</tbody>
</table>

### Single-threaded process

- Thread
- Registers
- Stack

### Multithreaded process

- Threads
- Registers
- Stacks

**Threads in one process share code, data, files, ...**
Why threads?

- **Express concurrency**
  - Web server (multiple requests), Browser (GUI + network I/O + rendering), ...

  ```c
  for(;;) {
      struct request *req = get_request();
      create_thread(process_request, req);
  }
  ```

- **Efficient communication**
  - Using a separate process for each task can be heavyweight
Threads vs. Processes

- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- There can be more than one thread in a process, the first thread calls `main()` & has the process’s stack
- Inexpensive creation
- Inexpensive context switching
- Efficient communication
- If a thread dies, its stack is reclaimed

- A process has code/data/heap & other segments
- A process has at least one thread
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- Expensive creation
- Expensive context switching
- Interprocess communication can be expressive
- If a process dies, its resources are reclaimed & all threads die
Using threads

- Through thread library
  - E.g. pthread, Win32 thread

- Common operations
  - create/terminate
  - suspend/resume
  - priorities and scheduling
  - synchronization
Example pthread functions

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void **(*start_routine)(void*), void *arg);
  - Create a new thread to run start_routine on arg
  - thread holds the new thread’s id
  - Can be customized via attr

- int pthread_join(pthread_t thread, void **value_ptr);
  - Wait for thread termination, and retrieve return value in value_ptr

- void pthread_exit(void *value_ptr);
  - Terminates the calling thread, and returns value_ptr to threads waiting in pthread_join
pthread creation example

```c
void* thread_fn(void *arg)
{
    int id = (int)arg;
    printf("thread %d runs\n", id);
    return NULL;
}

int main()
{
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread_fn, (void*)1);
    pthread_create(&t2, NULL, thread_fn, (void*)2);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

One way to view threads: function calls, except caller doesn’t wait for callee; instead, both run concurrently.

$ gcc –o threads threads.c –Wall –lpthread
$ threads
thread 1 runs
thread 2 runs
Outline

- Thread definition
- Multithreading models
- Synchronization
Multithreading models

- Where to support threads?

- **User threads**: thread management done by user-level threads library; kernel knows nothing

- **Kernel threads**: threads directly supported by the kernel
  - Virtually all modern OS support kernel threads
User vs. Kernel Threads

Example from Tanenbaum, Modern Operating Systems 3 e,
(c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639
User vs. Kernel Threads (cont.)

- Pros: fast, no system call for creation, context switch
- Cons: kernel doesn’t know ➔ one thread blocks, all threads in the process blocks

- Cons: slow, kernel does creation, scheduling, etc
- Pros: kernel knows ➔ one thread blocks, schedule another

No free lunch!
Multiplexing User-Level Threads

- A thread library must map user threads to kernel threads

- Big picture:
  - kernel thread: physical concurrency, how many cores?
  - User thread: application concurrency, how many tasks?

- Different mappings exist, representing different tradeoffs
  - Many-to-One: many user threads map to one kernel thread, i.e. kernel sees a single process
  - One-to-One: one user thread maps to one kernel thread
  - Many-to-Many: many user threads map to many kernel threads
Many-to-One

- Many user-level threads map to one kernel thread

**Pros**
- Fast: no system calls required
- Portable: few system dependencies

**Cons**
- No parallel execution of threads
  - All thread block when one waits for I/O
One-to-One

- One user-level thread maps to one kernel thread

- **Pros:** more concurrency
  - When one blocks, others can run
  - Better multicore or multiprocessor performance

- **Cons:** expensive
  - Thread operations involve kernel
  - Thread need kernel resources
Many-to-Many

- Many user-level threads map to many kernel threads ($U \geq K$)
  - Supported some versions of BSD, and Windows

- Pros: **flexible**
  - OS creates kernel threads for physical concurrency
  - Applications creates user threads for application concurrency

- Cons: **complex**
  - Most programs use 1:1 mapping anyway
Two-level

- Similar to M:M, except that a user thread may be bound to kernel thread
Other thread design issues

- Semantics of `fork()` system calls
  - Does `fork()` duplicate only the calling thread or all threads?
    - Running threads? Threads trapped in system call?
  - Linux `fork()` copies only the calling thread

- Signal handling
  - Which thread to deliver signals to?
  - Segmentation fault kills process or thread?
Problem:

- Creating a thread for each request: costly
  - And, the created thread exits after serving a request
- More user request → More threads, server overload

Solution: thread pool

- Pre-create a number of threads waiting for work
- Wake up thread to serve user request --- faster than thread creation
- When request done, don’t exit --- go back to pool
- Limits the max number of threads
Outline

- Thread definition
- Multithreading models
- Synchronization
int balance = 0;
int main()
{
    pthread_t t1, t2;
pthread_create(&t1, NULL, deposit, (void*)1);
pthread_create(&t2, NULL, withdraw, (void*)2);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
printf("all done: balance = %d\n", balance);
    return 0;
}

void* deposit(void *arg)
{
    int i;
    for(i=0; i<1e7; ++i)
        ++ balance;
}

void* withdraw(void *arg)
{
    int i;
    for(i=0; i<1e7; ++i)
        -- balance;
}
Results of the banking example

$ gcc -Wall -lpthread -o bank bank.c
$ bank
all done: balance = 0
$ bank
all done: balance = 140020
$ bank
all done: balance = -94304
$ bank
all done: balance = -191009

Why?
A closer look at the banking example

$ objdump --d bank
...
08048464 <deposit>:
...
0848473: a1 80 97 04 08     mov    0x8049780,%eax
0848478: 83 c0 01            add    $0x1,%eax
084847b: a3 80 97 04 08     mov    %eax,0x8049780
...

0804849b <withdraw>:
...
08484aa: a1 80 97 04 08     mov    0x8049780,%eax
08484af: 83 e8 01            sub    $0x1,%eax
08484b2: a3 80 97 04 08     mov    %eax,0x8049780
...
One possible schedule

CPU 0

mov 0x8049780,%eax
add $0x1,%eax
mov %eax,0x8049780

balance: 0
eax0: 0
eax0: 1
balance: 1

CPU 1

mov 0x8049780,%eax
add $0x1,%eax
mov %eax,0x8049780

balance: 0
eax1: 1
sub $0x1,%eax
eax1: 0
mov %eax,0x8049780

balance: 0

One deposit and one withdraw, balance unchanged. Correct
Another possible schedule

CPU 0

mov 0x8049780,%eax
add $0x1,%eax
mov %eax,0x8049780

CPU 1

balance: 0
eax0: 0

eax0: 1

mov 0x8049780,%eax

eax1: 0

balance: 1

eax1: -1

mov %eax,0x8049780

balance: -1

One deposit and one withdraw, balance becomes less. Wrong!
Race condition

- **Definition:** a timing dependent error involving shared state

- **Can be very bad**
  - “non-deterministic:” don’t know what the output will be, and it is likely to be different across runs
  - Hard to detect: too many possible schedules
  - Hard to debug: “heisenbug,” debugging changes timing so hides bugs (vs “bohr bug”)

How to avoid race conditions?

- **Atomic operations**: no other instructions can be interleaved, executed “as a unit” “all or none”, guaranteed by hardware.

- A possible solution: create a super instruction that does what we want atomically.
  - add $0x1, 0x8049780

- Problem
  - Can’t anticipate every possible way we want atomicity
  - Increases hardware complexity, slows down other instructions

```c
// ++ balance
mov 0x8049780,%eax
add $0x1,%eax
mov %eax,0x8049780
...

// -- balance
mov 0x8049780,%eax
sub $0x1,%eax
mov %eax,0x8049780
...```
Layered approach to synchronization

- Hardware provides simple low-level atomic operations, upon which we can build high-level, synchronization primitives, upon which we can implement critical sections and build correct multi-threaded/multi-process programs.
Example synchronization primitives

- **Low-level atomic operations**
  - On uniprocessor, disable/enable interrupt
  - On x86, aligned load and store of words
  - Special instructions:
    - test-and-set (TSL), compare-and-swap (XCHG)

- **High-level synchronization primitives**
  - Lock
  - Semaphore
  - Monitor