CSE 120
Principles of Operating Systems
Spring 2023
Lecture 7: Threads, Synchronization Intro
Yiying Zhang
Administrivia

- Homework 1 due and Homework 2 out tomorrow
#include <stdio.h>

void main()
{
    int pid; int was = 3;
    pid = fork(); /* fork another process */

    if (pid == 0) {/* child process */
        sleep(2); printf("child: was = %d\n", was);
        execlp("/bin/ls", "ls", NULL);
    } else {/* pid > 0; parent process */
        was = 4;
        printf("parent: child process id = %d; was=%d\n", pid, was);
        wait(NULL); exit(0);
    }
}

What are the possible print sequences of this program?
What are the possible print sequences of this program?

```c
#include <stdio.h>
void main()
{
    int pid;    int was = 3;
    pid = fork(); /* fork another process */

    if (pid == 0) { /* child process */
        sleep(2);  printf("child: was = %d\n", was);
        execlp("/bin/ls", "ls", NULL);
    } else { /* pid > 0; parent process */
        was = 4;
        wait(NULL);
        printf("parent: child process id = %d; was=%d\n", pid, was);
        exit(0);
    }
}
```
Processes

- Recall that a process includes many things
  - An address space (defining all the code and data pages)
  - OS resources (e.g., open files) and accounting information
  - Execution state (PC, SP, regs, etc.)

- Creating a new process is costly because of all of the data structures that must be allocated and initialized
  - Recall `task_struct` in Linux

- Communicating between processes is also costly
  - How to communicate? Each process is an island
  - The OS needs to intervene to bridge the gap
  - OS provides system calls to support Inter-Process Communication (IPC)
How do processes communicate?

- At process creation time
  - Parents get one chance to pass everything at fork()
- OS provides generic mechanisms to communicate
  - Shared Memory: multiple processes can read/write same physical portion of memory; implicit channel
    » System call to declare shared region
    » No OS mediation required once memory is mapped
  - Message Passing: explicit communication channel provided through send()/receive() system calls
    » A system call is required
- IPC is, in general, expensive due to the need for system calls
  - Although many OSes have various forms of lightweight IPC
Concurrent Programs

• Applications often want to execute several tasks in parallel
  ♦ When executing on a multi-core system, can get better perf
  ♦ These parallel tasks likely need to access some shared data and/or communicate with each other during their execution

• To execute these programs with processes, we need to
  ♦ Create several processes that execute in parallel (fork)
  ♦ Have each of them map a shared-data memory region (MMAP_SHARED)

• This situation is very inefficient
  ♦ **Space**: PCB, page tables, etc.
  ♦ **Time**: create data structures, fork and copy addr space, etc.
Rethinking Processes

• For some cases, forked processes are cooperative
  ♦ They all share the same code and data (address space)
  ♦ They all share the same privileges
  ♦ They all share the same resources (files, sockets, etc.)

• What don’t they share?
  ♦ Each has its own execution state: PC, SP, and registers

• Key idea: Why don’t we separate the concept of a process from its execution state?
  ♦ Process: address space, privileges, resources, etc.
  ♦ Execution state: PC, SP, registers

• Exec state also called thread of control, or thread
Threads

- Modern OSes (Windows, Unix, OS X) separate the concepts of processes and threads
  - The thread defines a sequential execution stream within a process (PC, SP, registers)
  - The process defines the address space and general process attributes (everything but threads of execution)
- A thread is bound to a single process
  - Processes, however, can have multiple threads
- Threads become the basic unit of scheduling
  - Processes are now the containers in which threads execute
Single and Multithreaded Processes

Single-threaded

Multithreaded
Threads in a Process

- Stack (T1)
- Stack (T2)
- Stack (T3)
- Heap
- Static Data
- Code

Thread 1
Thread 2
Thread 3
PC (T1)
PC (T2)
PC (T3)
Process/Thread Separation

- Separating threads and processes makes it easier to support parallel applications
  - Concurrency does not require creating new processes
- Concurrency (multithreading) can be very useful
  - Improving program structure
  - Handling concurrent events
  - Writing parallel programs
- So multithreading is even useful on a uniprocessor
  - Although today even cell phones are multicore
• **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
Threads: Concurrent Servers

- Using fork() to create new processes to handle requests in parallel is sometimes overkill
- Recall our forking Web server:

```c
while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
        Handle client request
        Close socket and exit
    } else {
        Close socket
    }
}
```
Threads: Concurrent Servers

- Instead, we can create a new thread for each request

```c
web_server() {
    while (1) {
        int sock = accept();
        thread_fork(handle_request, sock);
    }
}
```

```c
handle_request(int sock) {
    Process request
    close(sock);
}
```

Why would you still choose multi-processes to handle web requests?
Thread Design Space

- **One Thread per Process**
  - **One Address Space**
    - (MSDOS)
  - **Many Address Spaces**
    - (Early Unix)

- **Many Threads per Process**
  - **One Address Space**
    - (Java VM)
  - **Many Address Spaces**
    - (Solaris, Linux, NT, MacOS)
Threads Summary

- Many applications demand internal parallelism
  - Efficient parallelism requires fast primitives
  - Processes are too heavyweight
- Solution is to separate threads from processes
- Now, how do we get our threads (and processes with shared memory) to correctly cooperate with each other?
  - Synchronization…
Shared Resources

We first focus on coordinating access to shared resources

- **Basic problem**
  - If two concurrent threads (processes) are accessing a shared variable, and that variable is read/modified/written by those threads, then access to the variable must be controlled to avoid erroneous behavior

- **Over the next few lectures, we will look at**
  - Mechanisms to control access to shared resources
    - Locks, mutexes, semaphores, condition variables, etc.
  - Patterns for coordinating accesses to shared resources
    - Producer-consumer, reader-writer, etc.
Classic Example

• Suppose we have to implement a function to handle withdrawals from a bank account:

```c
withdraw (account, amount) {
    int balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

• Now suppose that you and your partner share a bank account with a balance of $1000

• Then you each go to separate ATM machines and simultaneously withdraw $100 from the account
Example Continued

• We’ll represent the situation by creating a separate thread for each person to do the withdrawals
• These threads run on the same bank server:

```c
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

• What’s the problem with this implementation?
  ♦ Think about potential schedules of these two threads
Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:

  ```
  balance = get_balance(account);
  balance = balance - amount;
  balance = get_balance(account);
  balance = balance - amount;
  put_balance(account, balance);
  put_balance(account, balance);
  ```

- What is the balance of the account now?
- Is the bank happy with our implementation?
- This problem is known as a data race
Shared Resources

- The problem is that two concurrent threads (or processes) accessed a shared resource (account) without any synchronization.
- We need mechanisms to control access to these shared resources in the face of concurrency.
  - So we can reason about how the program will operate.
- Our example was updating a shared bank account.
- Also necessary for synchronizing access to any shared data structure.
  - Buffers, queues, lists, hash tables, etc.
When Are Resources Shared?

• Local variables are not shared (private)
  ♦ Refer to data on the stack
  ♦ Each thread has its own stack

• Global variables and static objects are shared
  ♦ Stored in the static data segment, accessible by any thread

• Dynamic objects and other heap objects are shared
  ♦ Allocated from heap with malloc/free or new/delete
How contorted can the interleavings be?

- We'll assume that all instructions are atomic (either succeed completely or fail completely)
  - e.g., reads and writes of words
- We'll assume that a context switch can occur at any time
  - Examples may show code
  - But actually at instruction granularity
  - With multiprocessor, two insts can truly execute at the same time
- We'll assume that you can delay a thread as long as you like as long as it's not delayed forever

```c
get_balance(account);
balance = get_balance(account);
balance = balance - amount;
balance = balance - amount;
balance = get_balance(account);
balance = ...................................
put_balance(account, balance);
put_balance(account, balance);
```
Goal

• Make a larger chunk of code non-interleavable

• Let’s first understand the goal before discussing how to achieve it
Mutual Exclusion

• We can use **mutual exclusion** to synchronize access to shared resources
  ♦ Only one thread can access shared resources at a time
  ♦ This allows us to have larger atomic blocks

• Code block that uses mutual exclusion to synchronize its execution is called a **critical section**
  ♦ Only one thread at a time can execute in the critical section
  ♦ All other threads are forced to wait on entry
  ♦ When a thread leaves a critical section, another can enter
  ♦ Example: bathrooms on airplanes

• What requirements would you place on a critical section?
Mutual Exclusion Using Critical Sections

A enters critical region

A leaves critical region

B attempts to enter critical region

B blocked

B enters critical region

B leaves critical region

Time
Critical Section Goals

1) Mutual exclusion
   ♦ If one thread is in the critical section, then no other is

2) Progress
   ♦ If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
   ♦ A thread in the critical section will eventually leave it

3) Bounded waiting (no starvation)
   ♦ If some thread T is waiting on the critical section, then T will eventually enter the critical section

4) Performance
   ♦ The overhead of entering and exiting the critical section is small with respect to the work being done within it
About Goals

Also expressed as three properties:

- **Safety property**: nothing bad happens
  - Mutual exclusion
- **Liveness property**: something good happens
  - Progress, Bounded Waiting
- **Performance property**
  - Performance
- Rule of thumb: When designing a concurrent algorithm, worry about safety first (but don't forget liveness!). Performance is nice to have but won't affect correctness.
Mechanisms For Building Critical Sections

- **Locks**
  - Primitive, minimal semantics, used to build others
- **Semaphores**
  - Basic, easy to get the hang of, but harder to program with
- **Monitors / Conditional Variables**
  - High-level, requires language support, operations implicit
- **Messages**
  - Simple model of communication and synchronization based on atomic transfer of data across a channel
  - Direct application to distributed systems
### “Too Much Milk” Problem

<table>
<thead>
<tr>
<th>Roommate A</th>
<th>Roommate B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Look in fridge: out of milk</td>
<td>Look in fridge: out of milk</td>
</tr>
<tr>
<td>Leave for Ralphs</td>
<td>Leave for Ralphs</td>
</tr>
<tr>
<td>Arrive at Ralphs</td>
<td>Arrive at Ralphs</td>
</tr>
<tr>
<td>Buy milk</td>
<td>Buy milk</td>
</tr>
<tr>
<td>Arrive home</td>
<td>Arrive home</td>
</tr>
</tbody>
</table>

- How to enforce mutual exclusion?
A Possible Solution?

- Process can get context switched after checking milk and note, but before leaving note
- Why does it work for human?

```java
if ( no Milk ) {
    if (no Note) {
        leave note;
        buy milk;
        remove note;
    }
}
```

```java
if ( no Milk ) {
    if (no Note) {
        leave note;
        buy milk;
        remove note;
    }
}
```
Why does it work for people?

- Human can perform *test* (look for other person & milk) and *set* (leave note) at the same time.

- What we want is:

```java
Acquire(lock);
if (noMilk)
    buy milk;
Release(lock);
```
Locks

- A lock is an object in memory providing two operations
  - acquire() (or lock()): to enter a critical section
  - release() (or unlock()): to leave a critical section
- Threads pair calls to acquire and release
  - Between acquire/release, the thread holds the lock
  - acquire does not return until any previous holder releases
  - What can happen if the calls are not paired?
“Too much milk” problem with locks

acquire(lock);
if (no Milk)
    buy milk;
release(lock);

} Critical Section
Using Locks

withdraw (account, amount) {
  acquire(lock);
  balance = get_balance(account);
  balance = balance – amount;
  put_balance(account, balance);
  release(lock);
  return balance;
}

- What happens when blue tries to acquire the lock?
- Why is the “return” outside the critical section? Is this ok?
- What happens when a third thread calls acquire?
Next time...

• Read Chapters 30, 31

• Work on your project 1!