CS 3723 Operating Systems: Final Review

Instructor: Dr. Tongping Liu

Outline
- Threads
- Synchronizations
- Pthread Synchronizations

Threads: Outline
- Motivation and thread basics
  - Resources requirements: thread vs. process
- Thread implementations
  - User threads: e.g., Pthreads and Java threads
  - Kernel threads: e.g., Linux tasks
  - Map user- and kernel-level threads
  - Lightweight process and scheduler activation
- Other issues with threads: process creation and signals etc.
- Threaded programs
  - Thread pool
  - Performance vs. number of threads vs. CPUs and I/Os

Context Switches of Processes: Expensive
- Context switch between processes
  - Save processor registers for current process
  - Load the new process’s registers
- Switch address spaces – expensive
  - Hardware cache
  - Memory pages (e.g., TLB content)

Thread vs. Process
- Responsiveness
  - Part of blocked
- Resource Sharing
  - Memory, open files, etc.
- Economy
  - Creation and switches
- Scalability
  - Increase parallelism

Process: Alternative View
- Process = thread + code, data, and kernel context
Process with Two Threads

Thread 1
- Program context:
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)
- Code, data, and kernel context
  - Shared libraries
  - Run-time heap
  - Read/write data
  - Read-only code/data

Thread 2
- Program context:
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)
- Kernel context:
  - VM structures
  - Descrptor table
  - Brk pointer

Resources for Threads

- **Shared** resources among threads (per process items)
  - Address space (e.g., codes)
  - Global variables/data
  - Open Files and other resources etc.

- **Separated** resources for each thread
  - Machine state: registers (e.g., PC)
  - Running stacks
  - Private data

Threads vs. Processes

- **Threads and processes: similarities**
  - Each has its own logical control flow
  - Each can run concurrently with others
  - Each is context switched (scheduled) by the kernel

- **Threads and processes: differences**
  - Threads share code and data, processes (typically) do not
  - Threads are less expensive than processes
    - Process control (creation and exit) is more expensive as thread control
    - Context switches: processes are more expensive than for threads

Pros and Cons of Thread-Based Designs

- **Pros**
  - Easy to share data structures between threads
    - E.g., logging information, file cache
  - Threads are more efficient than processes

- **Cons**
  - Unintentional sharing can introduce subtle and hard-to-reproduce errors!

Many-to-One Model

- **Pros:**
  - Cheap synchronization and cheap thread creation

- **Cons:**
  - Blocking-problem. A thread calling block system call will block the whole process
  - No concurrency.

One-to-one Model

- **Pros:**
  - Scalable parallelism (concurrency)
  - Thread will not block a whole process

- **Cons:**
  - Expensive synchronization (system call is required if a lock can't be acquired)
  - Expensive creation (3.5 slower)
  - Kernel resource, e.g., stack and kernel structure
Many-to-Many Model

Pros:
- Cheap Resource, not all user threads should create a kernel thread
- Synchronization mainly at user-level
- Context switch may not involve system calls

Cons:
- Difficult cooperation between kernel scheduler and user scheduler
- How to decide the number of kernel threads?

Pthreads: POSIX Thread

- POSIX
  - Portable Operating System Interface (POSIX)
  - Standardized programming interface

- Pthreads
  - Thread implementations adhering to POSIX standard
  - API specifies behavior of the thread library: defined as a set of C types and procedure calls
  - Common in UNIX OS (Solaris, Linux, Mac OS X)
  - Support for thread creation and synchronization

Shared Variables in Threaded C Programs

- Question: Which variables in a threaded C program are shared?
  - The answer is not as simple as "global variables are shared" and "stack variables are private"

- Requires answers to the following questions:
  - What is the memory model for threads?
  - How are variables mapped to memory?
  - How many threads might reference each variable?

- A variable \(x\) is shared if and only if multiple threads reference some instance of \(x\)

Threads Memory Model

- Conceptual model:
  - Multiple threads run in the same context of a process
  - Each thread has its own separate thread context
  - Thread ID, stack pointer, PC, and GP registers
  - All threads share the remaining process context
    - Code, data, heap, and shared library segments
    - Open files and installed handlers

- Operationally, this model is not strictly enforced:
  - Register values are truly separate and protected, but...
  - Any thread can read and write the stack of any other thread

Mapping Variable Instances to Memory

- Global variables
  - Def: Variable declared outside of a function
  - Virtual memory contains exactly one instance of any global variable

- Local variables
  - Def: Variable declared inside function without static attribute
  - Each thread stack contains one instance of each local variable

- Local static variables
  - Def: Variable declared inside function with the static attribute
  - Virtual memory contains exactly one instance of any local static variable.
Thread Pool

- Pool of threads
  - Threads in a pool where they wait for work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
- Adjust thread number in pool
  - According to usage pattern and system load

Performance of Threaded Programs

- Suppose that the processing of each request
  - Takes X seconds for computation; and
  - Takes Y seconds for reading data from I/O disk
- For single-thread program/process
  - A single CPU & single disk system
  - What is the maximum throughput (i.e., the number of requests can be processed per second)?

Example: suppose that each request takes
2ms for computation
8ms to read data from disk
1000/10ms = 100

Adjust thread number in pool
According to usage pattern and system load

Performance of Threaded Programs (cont)

- Multi-thread performance improvement
  - Single CPU & single disk system
  - How many threads should be used?
  - What is the maximum throughput (i.e., the number of requests can be processed per second)?

Example: suppose that each request takes
2ms for computation
8ms to read data from disk
1000/8ms = 125
Assuming that we have 8 cores and 1 disk

Performance of Threaded Programs (cont)

- What about m-CPU and n-disk system
  - Maximum throughput and # of threads? (X: computation, Y: IO for each task)
  - Throughput \( \rightarrow \frac{1}{\max(X/m, Y/n)} \)
  - if \( (X/m < Y/n) \), \# = n + m'
    where \( m' = \min\{k| X/k \leq Y/n, 1 \leq k \leq m\} \);
  - Similarly, if \( (X/m > Y/n) \), \# = m + n'
    where \( n' = \min\{k| X/m' \geq Y/k, 1 \leq k \leq n\} \);
- Other issues:
  - When I/O disk is bottleneck, adding more CPUs will NOT help improve the throughput
  - What about heterogeneous disks and CPUs?!

Synchronization: Outline

- Memory Model of Multithreaded Programs
- Synchronization for coordinated processes/threads
- The produce/consumer and bounded buffer problem
- Critical Sections and solutions’ requirements
- Peterson’s solution for two processes/threads
- Synchronization Hardware: atomic instructions
- Semaphores
  - Two operations (wait and signal) and standard usages
  - Bounded buffer problem with semaphores
  - Initial values and order of operations

Concurrent Access to Shared Data

- Two threads A and B have access to a shared variable “Balance”
  - Thread A:
    Balance = Balance + 100
  - Thread B:
    Balance = Balance - 200

A1. LOAD R1, BALANCE
A2. ADD R1, 100
A3. STORE BALANCE, R1

B1. LOAD R1, BALANCE
B2. SUB R1, 200
B3. STORE BALANCE, R1
What is the problem then?

**Observe:** In a time-shared system, the *exact instruction execution order* cannot be predicted

- **Scenario 1:**
  1. LOAD R1, BALANCE
  2. ADD R1, 100
  3. STORE BALANCE, R1
  Context Switch!
  **Scenario 2:**
  1. LOAD R1, BALANCE
  2. ADD R1, 100
  3. STORE BALANCE, R1
  Context Switch!

*Sequential correct execution*
- Balance is effectively decreased by 100!

*Mixed wrong execution*
- Balance is effectively decreased by 200!!

Race Conditions

- Multiple processes/threads write/read *shared data* and the outcome depends on the particular order to access shared data are called *race conditions*.
  - A serious problem for concurrent system using shared variables!

  **How do we solve the problem?!**

- Need to make sure that some high-level code sections are executed *atomically*.
  - Atomic operation means that it completes in its entirety without worrying about interruption by any other potentially conflict-causing process.

What is Synchronization?

- Cooperating processes/threads share data & have effects on each other → executions are NOT reproducible with non-deterministic exec. speed.

- Concurrent executions
  - Single processor → achieved by time slicing
  - Parallel/distributed systems → truly simultaneous

- Synchronization → getting processes/threads to work together in a *coordinated* manner.

Mutual Exclusion

- **Critical Section (CS)**
  - A section of code that modify the *same shared* variables/data that must be executed *mutually exclusively* in time.

- **General structure for processes/threads with CS**
  - *entry section:* The code which requests permission to enter the critical section.
  - *critical section:* as above
  - *exit section:* The code which removes the mutual exclusion.
  - *remainder section:* Everything else

General Structure for Critical Sections

```c
    do {
        __entry_section
        __critical_section
        __exit_section
        __remainder_section
    } while (1);
```

- In the *entry section*, the process requests "permission".

Critical Sections
Requirements for CS Solutions

- Mutual Exclusion
  - At most one process/thread in its CS at any time

- Progress
  - If all other processes/threads are in their remainder sections, a process/thread is allowed to enter its CS
  - Only those processes that are not in their remainder section can decide which process can enter its CS next, and this decision cannot be postponed indefinitely.

- Bounded Waiting
  - Once a process has made a request to enter its CS, other processes can only enter their CSs with a bounded number of times

A Simple Solution

- int turn; → indicate whose turn to enter CS
- T0 and T1: alternate between CS and remainder

Peterson’s Solution

- Three shared variables: turn and flag[2]

Hardware Support for Synchronization

- Synchronization
  - Need to test and set a value atomically

- IA32 hardware provides a special instruction: xchg
  - When the instruction accesses memory, the bus is locked during its execution and no other process/thread can access memory until it completes!!!
  - Other variations: xchgb, xchgw, xchg1

Hardware Instruction TestAndSet

- The TestAndSet instruction tests and modifies the content of a word atomically (non-interruptable)

- Keep setting the lock to 1 and return old value.

- Swap contents of two memory words

- What’s the problem?
  1. Busy-waiting, waste cpu
  2. Hardware dependent, not bounded-waiting
Semaphores

- Synchronization without busy waiting
  - Motivation: Avoid busy waiting by blocking a process execution until some condition is satisfied
- Semaphore S — integer variable
  - Two indivisible (atomic) operations:
    - wait(s) (also called P(s) or down(s) or acquire());
    - signal(s) (also called V(s) or up(s) or release())
  - User-visible operations on a semaphore
  - Easy to generalize, and less complicated for application programmers

Semaphore Usage

- Counting semaphore —
  - Can be used to control access to a given resources with finite number of instances
- Binary semaphore — integer value can range only between 0 and 1; Also known as mutex locks

Semaphore for General Synchronization

- Execute code B in Pj after code A is executed in Pi
- Use semaphore flag: what is the initial value?
- Code
  ```
P_i:  P_j:  
  ...  ...
  wait(mutex);  wait(mutex);  
  ...  ...
  critical section  critical section  
  signal(mutex);  signal(mutex);  
  remainder section  remainder section  
  signal(flag);  signal(flag);  
  ...
  ...
```

Attacking CS Problem with Semaphores

- Shared data
  - semaphore mutex = 1; /* initially mutex = 1 */
- For any process/thread
  ```
do {  
  ...  
  wait(mutex);  
  critical section  
  signal(mutex);  
  remainder section  
  ↙
} while(1);
```

Producer-Consumer Code using Semaphore

```java
Producer

do {  
  produce an item in nextp  
  ...
  wait(empty);  
  wait(mutex);  
  ...
  add nextp to buffer  
  ...
  signal(mutex);  
  signal(full);  
} while (1)

Consumer

do {  
  wait(full);  
  wait(mutex);  
  ...
  remove an item from buffer to nextc  
  ...
  signal(mutex);  
  signal(empty);  
  ...
  consume the item in nextc  
  ...
} while (1)
```

Outline of Pthreads Synchronization

- High-level synchronization structure: Monitor
- Pthread mutex
- Conditional variables
- Barrier
- Threading Issues
monitors vs. semaphores

- A Monitor:
  - An object designed to be accessed across threads
  - Member functions enforce mutual exclusion

- A Semaphore:
  - A low-level object
  - We can use semaphore to implement a monitor

Pthreads: Thread/Synchronization APIs

<table>
<thead>
<tr>
<th>Thread Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create</td>
<td>Create a new thread in the caller’s address space</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>Terminate the calling thread</td>
</tr>
<tr>
<td>pthread_join</td>
<td>Wait for a thread to terminate</td>
</tr>
<tr>
<td>pthread_mutex_init</td>
<td>Create a mutex</td>
</tr>
<tr>
<td>pthread_mutex_destroy</td>
<td>Destroy a mutex</td>
</tr>
<tr>
<td>pthread_mutex_lock</td>
<td>Lock a mutex</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>Unlock a mutex</td>
</tr>
<tr>
<td>pthread_cond_init</td>
<td>Create a condition variable</td>
</tr>
<tr>
<td>pthread_cond_destroy</td>
<td>Destroy a condition variable</td>
</tr>
<tr>
<td>pthread_cond_wait</td>
<td>Wait on a condition variable</td>
</tr>
<tr>
<td>pthread_cond_signal</td>
<td>Release one thread waiting on a condition variable</td>
</tr>
</tbody>
</table>

Semaphore vs. Mutex_lock

- Definition and initialization
  ```c
  volatile int cnt = 0;
  sem_t mutex = 1;
  // Initialize to Unlocked
  pthread_mutex_init(&mutex, NULL);
  ```

- Entering and Exit CS
  ```c
  for (i = 0; i < niters; i++) {
    Wait(&mutex);
    cnt++;
    Signal(&mutex);
  }
  ```

Binary Semaphore and Mutex Lock?

- Binary Semaphore:
  - No ownership

- Mutex lock
  - Only the owner of a lock can release a lock.
  - Priority inversion safety: potentially promote a task
  - Deletion safety: a task owning a lock can’t be deleted.

Condition Variables

- Special `pthread` data structure
  - Make it possible/easy to go to sleep
    - Atomically:
      - release lock
      - put thread on wait queue
      - go to sleep
  - Each CV has a queue of waiting threads
  - Do we worry about threads that have been put on the wait queue but have NOT gone to sleep yet?
    - no, because those two actions are atomic
  - Each condition variable associated with one lock

Condition Variable vs. Semaphore

```c
void * consumer_dequeue() {
  void * item = NULL;
  pthread_mutex_lock(&l);
  while (q.empty()) {
    pthread_cond_wait(&nempty, &l);
  }
  item = q.pop_back();
  pthread_cond_signal(&nfull);
  pthread_mutex_unlock(&l);
  return item;
}
```
**Barrier**

```
<table>
<thead>
<tr>
<th>Time</th>
<th>Epoch0</th>
<th>Epoch1</th>
<th>Epoch2</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Thread&quot; 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;Thread&quot; 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;Thread&quot; 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Lock Problems**
- Race condition
- Atomicity violation
- Order violation
- Deadlock

**Issues Related to Conditional Variables**
- `signal()` before `wait()`
  - Waiting thread may miss the signal
- Fail to lock mutex before `wait`
  - May return error, or not blocking
- `if (!condition) wait();` instead of `while (!condition) wait();`
  - Condition may still fail when waken up
  - May lead to arbitrary errors, such as segmentation fault
- Forgot to unlock mutex after signal/wakeup

https://courses.engr.illinois.edu/cs241/sp2014/lecture/22-condition-deadlock.pdf

**Overall**
- Process: concept, memory model, scheduling, basic programing (`fork()`, `wait()`)
- File system: file, file pointer, links, FDT, SFT, inode
- IPC: pipe, fifo, dup2
- Memory management: paging, page table, TLB, buddy, page replacement
- Threads: concept, memory model, difference with processes, threading model, basic programing
- Synchronization: hardware instruction, semaphore
- Thread-Based Synchronization: lock, conditional variables, barrier, synchronization issue