CS 3723 Operating Systems:
Final Review

Instructor: Dr. Tongping Liu

Lecture Outline
- High-level synchronization structure: Monitor
- Pthread mutex
- Conditional variables
- Barrier
- Threading Issues

Monitors
- High-level synchronization construct (implement in different languages) that provided mutual exclusion within the monitor AND the ability to wait for a certain condition to become true

```
monitor monitor-name{
    shared variable declarations
    procedure body P1 (...) { . . . }
    procedure body P2 (...) { . . . }
    procedure body Pn (...) { . . . }

    {initialization codes; }
```

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

Semaphore vs. Mutex_lock
- Definition and initialization

```
volatile int cnt = 0;
sem_t mutex = 1;
```

- Entering and Exit CS

```
for (i = 0; i < niters; i++) {
    Wait(&mutex);
    cnt++;
    Signal(&mutex);
}
```

```
for (i = 0; i < niters; i++) {
    pthread_mutex_lock(&mutex);
    cnt++;
    pthread_mutex_unlock(&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.
Synchronization Operations

- **Safety**
  - Locks provide mutual exclusion
  - But, we need more than just mutual exclusion of critical regions...

- **Coordination**
  - *Condition variables* provide ordering

**Condition Variables**

- **Wait for 1 event, atomically release lock**
  - `wait(Lock& l, CV& c)`
    - If queue is empty, wait
    - Atomically releases lock, goes to sleep
    - You must be holding lock!
    - May reacquire lock when awakened (pthreads do)
  - `signal(CV& c)`
    - Insert item in queue
    - Wakes up one waiting thread, if any
  - `broadcast(CV& c)`
    - Wakes up all waiting threads

- **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;
}

void producer_enqueue(void *item){
    pthread_mutex_lock(&l);
    while (q.full()) {
        pthread_cond_wait(&nfull, &l);
    }
    q.push_front(item);
    pthread_cond_signal(&nempty);
    pthread_mutex_unlock(&l);
}
```

**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**

Outline

- **Client-server model**
  - Widely used in many applications: ftp, mail, http, ssh
- **Communication models**
  - Connectionless vs. connection-oriented
- **Network basics: TCP/IP protocols**
- **Connection-oriented Client-Server communication**
  - Sockets & usages: socket, bind, listen, accept, connect
- **UICI: A Simple Client/Server Implementation**
  - Simplified interface: u_open, u_accept, u_connect

Client-Server Model

- **Applications with two distinct parts**
  - Server vs. client
- **Server**: wait for requests from clients
- **Client**: makes requests for services from server
- **Examples:**
  - Web server vs. clients: http and FireFox, Chrome etc.
  - Mail server vs. clients
  - File server vs. client: nfs

Connection-oriented Client-Server Model

- **Client** sets up a connection to server’s well-known port number
  - After that, communicate over the private channel

Pros and Cons?

Many Clients vs. One Server

- **Clients** use the same passive endpoint initially

How does server handle each request? sequentially?

Server are typically powerful to handle multiple requests
Handle Concurrent Requests

- Server can fork a child process to handle each request from different clients
  - Too costly, use threads!
  - Apache, nginx: hybrid multi-process multi-threaded. Why?

![Diagram of server and client hosts with fork process]

Thread Pools

- Tasks are defined as Runnable of each Executor object
- Define in Execute() of the executor class

![Diagram of thread pool with tasks and execution policy]

Advantage of Thread Pools

- Reusing an existing thread; reduce thread creation and teardown costs.
- No latency associated with thread creation; improves responsiveness.

By properly tuning the size of the thread pool, you can have enough threads to keep the processors busy while not having so many that your application runs out of memory or thrashes due to competition among threads for resources.

Sizing Thread Pools

- The ideal size for a thread pool depends on the types of tasks and the deployment system
  - If it is too big, performance suffers
  - If it is too small, throughput suffers

- Heuristics
  - For compute intensive tasks, N+1 threads for a N-processor system
  - For tasks including I/O or other blocking operations, you want a larger pool

IP Solves the Routing Problem

- Decide the route for each packet
  - necessary in MANs and WANs
- Update knowledge of the network
  - Adaptive/dynamic routing is usually used: traffic patterns, topological changes
- Routing decision
  - hop-by-hop, with period update and distribution of traffic data, e.g., the distance-vector, dynamic, distributed algorithm

Programmer’s View of TCP/IP

- Transport layer: TCP vs. UDP
- TCP (Transmission Control Protocol):
  - connection-oriented stream service
- UDP (User Datagram Protocol):
  - connectionless datagram service
TCP: Transmission Control Protocol

- TCP is connection-oriented.
  - 3-way handshake used for connection setup
  - Acknowledge each message (piggyback)

 ![3-way handshake diagram](image)

Connection Setup
3-way handshake

- Acknowledgement data packets

Socket Abstraction

- A socket must be bound to a local port
- Provide endpoints for communication between processes
- Socket pair - (local IP address, local port, foreign IP address, foreign port) uniquely identifies a communication channel

TCP Socket Primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Socket</td>
<td>Create a new communication endpoint</td>
</tr>
<tr>
<td>Bind</td>
<td>Attach a local address to a socket</td>
</tr>
<tr>
<td>Listen</td>
<td>Announce willingness to accept connections</td>
</tr>
<tr>
<td>Accept</td>
<td>Block caller until a connection request arrives</td>
</tr>
<tr>
<td>Connect</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>Send</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>Recv</td>
<td>Receive some data over the connection</td>
</tr>
<tr>
<td>Close</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>

Client-Server Using TCP Sockets

- Server side performs first with following actions
  - Step 1: Create socket
  - Step 2: bind server address & port (# known to clients)
  - Step 3: start listen for a connection request
  - Step 4: accept a connection request → create another socket for private communication with the clients
- Client: create a socket with (IP/host-name, port#), and then try to connect to the server

Programming with UDP/IP sockets

1. Create the socket
2. Identify the socket (name it)
3. On the server, wait for a message
4. On the client, send a message
5. Send a response back to the client (optional)
6. Close the socket

No need to setup the channel

Lecture 1: Introduction

- Operating System: what is it?
- Evolution of Computer Systems and OS Concepts
- Different types/variations of Systems/OS
  - Parallel/distributed/real-time/embedded OS etc.
- OS as a resource manager
  - How does OS provide service? – interrupt/system calls
- OS Structures and basic components
  - Process/memory/I/O device managers

OS definition, components and system calls
Lecture 2: Programs and Processes
- Programs, Processes and Threads
- Process creation and its components
- States of a process and transitions
- PCB: Process Control Block
- Process (program image) in memory
- Argument Arrays
- Storage and Linkage Classes
- Creation and Exit of Processes

Lecture 3-4: Processes and Scheduling
- Reviews on process
- Process queues and scheduling events
- Different levels of schedulers
- Preemptive vs. non-preemptive
- Context switches and dispatcher
- Performance criteria
  - Fairness, efficiency, waiting time, response time, throughput, and turnaround time;
- Classical schedulers: FIFO, SJF, PSFJ, and RR
- CPU Gantt chart vs. process Gantt charts

Lecture 5: File System
- Basics of File Systems
- Directory and Unix File System: inodes
- UNIX I/O System Calls: open, close, read, write, ioctl
- File Representations: FDT, SFT, inode table
- fork and inheritance, Filters and redirection
- File pointers and buffering
- Directory operations
- Links of Files: Hard vs. Symbolic

Lecture 6: IPC
- Inter-Process communication (IPC)
- Pipe and its operations
- FIFOs: named pipes
  - Allow un-related processes to communicate
- Ring of communicating processes
  - Steps for Ring Creation with Pipes
  - A Ring of n Processes
- Other issues in token ring
  - Which process to write/read: token management

Lecture 07: Memory Management
- Simple memory management: swap etc.
- Virtual memory and paging
  - Page table and address translation
  - Translation lookaside buffer (TLB)
- Multi-level page table
- Page replacement algorithms and modeling
- Working set of processes

Lecture 08: Memory Management
- Virtual Memory
- Demand Paging
- Physical Memory Management (Buddy and Slap)
- Page Replacement
- Memory Mapping (mmap)
- Connecting with User Space Memory
Lecture 11: Threads

- 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

Lecture 12: Synchronization

- 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

Lecture 13: Thread Synchronization

- High-level synchronization structure: Monitor
- Pthread mutex
- Conditional variables
- Barrier
- Threading Issues

Lecture 14: Network Communication

- Client-server model
  - Widely used in many applications: ftp, mail, http, ssh
- Communication models
  - Connectionless vs. connection-oriented
- Network basics: TCP/IP protocols
- Connection-oriented Client-Server communication
  - Sockets & usages: socket, bind, listen, accept, connect
- UICI: A Simple Client/Server Implementation
  - Simplified interface: u_open, u_accept, u_connect

Overall

- Process: concept, memory model, scheduling, basic programming (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 programming
- Synchronization: hardware instruction, semaphore
- Thread-Based Synchronization: lock, conditional variables, barrier, synchronization issue
- Network Communication: connection-less and connect-oriented