CS 5523: Operating Systems

Instructor: Dr. Tongping Liu

Final Reviews (Comprehensive)

Final Exam: May 5, 2015, Tuesday
6:00pm – 8:30pm
Lecture 1: OS Overview

- 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/IO device managers
- Basic design approaches
  - Monolithic/layered/microkernel/virtual machine etc.
Lecture 2: Process Management

- Basic concepts of process
  - Process control block (PCB) and address space

- Basic operations for process management
  - Process creation/termination

- States of process: different queues
  - ready, running, or wait etc
  - Context switch: multiple hardware running contexts

- Scheduling of process: CPU scheduling
  - Basic scheduling algorithms: FIFO, SJF etc

- Inter process communication
  - shared memory and message
Lecture03: Thread and Implementation

- 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
Lecture04: Concurrency and Synchronization

- Problems with concurrent access to shared data
  - Race condition and critical section
  - General structure for enforce critical section

- Synchronization mechanism
  - Hardware supported instructions: e.g., TestAndSet
  - Software solution: e.g., semaphore

- Classical Synchronization Problems

- High-level synchronization structure: Monitor

- Case study for synchronization
  - Pthread library: mutex and conditional variables
  - Java inherit monitor and conditional variable
Lecture 05: Memory Management

- Simple memory management: swap etc.
- Virtual memory and paging
  - Page table and address translation
- Translation lookaside buffer (TLB)
- Multi-level page table and inverted page table
- Track free memory: bitmaps or linked list
- Page replacement algorithms and modeling
- Working set of processes
- Other implementation issues
  - Dirty page and locking page etc.
Lecture06: Distributed Systems

- Distributed Systems: examples and definition
- OS structures in distributed systems
  - Distributed OS
  - Networked OS
  - Middleware Based systems
- Design objectives of distributed systems
  - Resource availability, transparency, openness and scalability
- Different distributed systems
- Misconceptions about distributed systems
Lecture07: Network Communication

- Layered network models
  - OSI 7-layer model (Open System Interconnection)
- Ethernet: local area network
- Inter-network Protocols (IP)
  - Addressing and routing etc.
- TCP/UDP protocols
  - communication ports and sockets
- Multicast: more than one recipients
Lecture08: Application-Level Communications

- **Fundamentals**
  - Client/Server communication protocols
    - Request vs. Request-reply vs. Request-reply-acknowledge
  - Invocation semantics
    - Exact once vs. at least once vs. at most once
  - Communication types
    - Transient vs. persistent
    - Synchronous vs. asynchronous

- **Models for application communications**
  - **RPC**: remote procedure call
  - Message-oriented communication
  - Stream-Oriented communication
  - Multicast communication
Lecture09: Remote Objects and RMI

- Distributed/Remote Objects
- Remote object reference (ROR)
- Remote Method Invocation (RMI)
- Case study and example: Java RMI
Lecture 10: Naming and Name Service

- Name and name services
  - Naming space and implementation

- Flat name and resolutions
  - Forwarding pointers
  - Distributed Hash Table (DHT): Chord

- Structure name
  - Name resolution: iterative vs. recursive
  - Case study: DNS

- Attributed-based naming
  - Directory service
  - Case study: X.500 and LDAP
Lecture 11: DS Synchronizations

- Physical clock/time in distributed systems
  - No global time is available
  - Network Time Protocol
  - Berkeley Algorithm

- Logical clock/time and ‘Happen Before’ Relation
  - Lamport’s logical clock → total ordering multicast
  - Vector clocks → Causally ordering

- Mutual Exclusion: Distributed synchronizations
  - De/Centralized algorithms
  - Distributed algorithms (Ricart & Agrawala)
  - Logical token ring
Computer Clocks and Timing Events

- Processes on different computers can timestamp their events using their own clocks
  - Clocks on different computers may give different times
  - Computer clocks drift from perfect time and their drift rates differ from one another
NTP: basic idea
At least one machine has a UTC receiver

- Suppose we have a server with UTC receiver.
- The server has an accurate clock
- So clients can simply contact it and get the accurate time (every $\delta/2\rho$ sec)

- A gets $T_1, T_2, T_3, T_4$.
- How should A adjust its clock?
- The problem is the delay which causes inaccuracy
NTP: basic idea
Suppose propagation delay is the same in both ways?

Assume \( dT_{req} = dT_{res} \)

A can estimate its offset value to B as \( \theta \)

\[
\theta = T_3 + \frac{(T_2-T_1)+(T_4-T_3)}{2} - T_4 = \frac{(T_3-T_4) + (T_2-T_1)}{2}
\]

Confuse: the object file is earlier than the source

\( \theta \geq 0 \), A is slower
\( \theta < 0 \), A is faster, but time cannot run backward?

Introduce the difference gradually (e.g., if time generate 100 interrupts, instead of 10ms, we add 9ms for each interrupt.)
Berkeley Algorithm
No machine has UTC receiver

- Time does not need to be the actual time…
- As long as all machines agree, then that is OK for many applications
- Gradually advance or slow down the clock…
Logical Time

- The order of two events occurring at two different computers cannot be determined based on their “local” time.

- **Problem**: How do we maintain a global view on the system’s behavior that is consistent with the happened-before relation.

- *The notion of logical time/clock is fairly general and constitutes the basis of many distributed algorithms*
“Happened Before” Relation

- Lamport first defined a “happened before” relation (→) to capture the causal dependencies between events.

- **Same process**: A → B, if A and B are events in the same process and A occurred before B.

- **Different processes**: A → B, if A is the event of sending a message m in a process and B is the event of the receipt of the same message m by another process.

- If A → B, and B → C, then A → C (happened-before relation is transitive).
“Happened Before”: Partial Order

- $a \rightarrow b$ (at $p_1$); $c \rightarrow d$ (at $p_2$); $b \rightarrow c$; also $d \rightarrow f$
- Not all events are related by the “$\rightarrow$” relation
  - $a$ and $e$ (different processes and no message chain)
  - They are not related by “$\rightarrow$”
  - They are said to be concurrent (written as $a \parallel e$)
Logical Clock: Example

<table>
<thead>
<tr>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>24</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td>30</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>36</td>
<td>48</td>
<td>60</td>
</tr>
<tr>
<td>42</td>
<td>56</td>
<td>70</td>
</tr>
<tr>
<td>48</td>
<td>64</td>
<td>80</td>
</tr>
<tr>
<td>54</td>
<td>72</td>
<td>90</td>
</tr>
<tr>
<td>60</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>24</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td>30</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>36</td>
<td>48</td>
<td>60</td>
</tr>
<tr>
<td>42</td>
<td>56</td>
<td>70</td>
</tr>
<tr>
<td>48</td>
<td>64</td>
<td>80</td>
</tr>
<tr>
<td>54</td>
<td>72</td>
<td>90</td>
</tr>
<tr>
<td>60</td>
<td>76</td>
<td></td>
</tr>
</tbody>
</table>

(b)

P_2 adjusts its clock

P_1 adjusts its clock

m_1
m_2
m_3
m_4
Observation: Lamport’s clocks do not guarantee that if $C(a) < C(b)$ that $a$ causally preceded $b$:

- Event $a$: $m_1$ is received at $T = 16$.
- Event $b$: $m_2$ is sent at $T = 20$.

We cannot conclude that $a$ causally precedes $b$. 
Vector Clocks

Vector clocks are constructed by letting each process $P_i$ maintain a vector $VC_i$ with the following two properties:

1. $VC_i[i]$ is the number of events that have occurred so far at $P_i$. In other words, $VC_i[i]$ is the local logical clock at process $P_i$.

2. If $VC_i[j] = k$ then $P_i$ knows that $k$ events have occurred at $P_j$. It is $P_i$’s knowledge of the local time at $P_j$. 

CS5523: Operating System @ UTSA
Vector Clocks: Update

- Rule 1: Before executing an event $P_i$, executes $VC_i[i] \leftarrow VC_i[i] + 1$.

- Rule 2: When process $P_i$ sends $m$ to $P_j$, it sets $m$’s (vector) timestamp $ts(m) = VC_i$ after Rule 1;

- Rule 3: Upon the receipt of $m$, process $P_j$ adjust $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each $k$, after which it executes Rule 1 and delivers $m$ to the application.

It is possible to ensure that a message is delivered only if all messages that causally precede it have been received.
Vector Clocks: Update

- Rule 1: Before executing an event $P_i$, executes $\text{VC}_i[i] \leftarrow \text{VC}_i[i] + 1$.

- Rule 2: When process $P_i$ sends $m$ to $P_j$, it sets $m$’s (vector) timestamp $ts(m) = \text{VC}_i$ after Rule 1;

- Rule 3: Upon the receipt of $m$, process $P_j$ adjusts $\text{VC}_j[k] \leftarrow \max\{\text{VC}_j[k], ts(m)[k]\}$ for each $k$, after which it executes Rule 1 and delivers $m$ to the application.

It is possible to ensure that a message is delivered only if all messages that causally precede it have been received.
Causally-Ordered Multicasting

- Ensure to deliver a message only if all causally preceding messages have already been delivered

- $P_j$ postpones delivery of $m$ from $P_i$ until:
  - $R1$: $ts(m)[i] = VC_j[i] + 1$;
  - $m$ is the next message expected from $P_i$

  - $R2$: $ts(m)[k] \leq VC_j[k]$ for $k \neq i$
  - $P_j$ see all messages that have been seen by $P_i$ when it sent out message $m$. 
For $P_2$, when receive $m^*$ from $P_1$, $\text{ts}(m^*) = (1,1,0)$, but $\text{VC}_2 = (0,0,0)$; \textit{m* is delayed as $P_2$ didn’t see message from $P_0$ before;}

Whe $P_2$ receive $m$ from $P_0$, $\text{ts}(m) = (1,0,0)$, with $\text{VC}_2 = (0,0,0)$ $\Rightarrow$ both R1 and R2 is ok, and $m$ is delivered $\Rightarrow$ $\text{VC}_2 = (1,0,0)$, then $m^*$ is delivered
Mutual Exclusion in Distributed Systems

To ensure exclusive access to some resource for processes in a distributed system

- Permission-based vs. token-based approaches

Solutions

- Centralized server;
- Decentralized, using a peer-to-peer system;
- Distributed, with no topology imposed;
- Completely distributed along a (logical) ring;
Lecture 12: Consistency & Replication

- Motivations for replications
  - Performance and/or fault-tolerance

- Data-Centric Consistency Models
  - Continuous Consistency, Consistent Ordering of Operations

- Client-Centric Consistency Models
  - Eventual Consistency
  - Monotonic Reads, Monotonic Writes
  - Read Your Writes, Writes Follow Reads

- Replica Management
  - Replica-Server Placement, Content Replication & Placement
  - Content Distribution

- Consistency Protocols
  - Implementation of the consistency models
Why Replications are Needed?

- Data are replicated
  - To increase the reliability of a system:
    - If one crash, we can switch to another one
    - Provide better protection on the data
  - To improve performance → **Scalability**
    - Scaling in numbers and in geographical area (e.g., place copies of data close to the processes using them. So clients can quickly access the content.)

- Problems
  - How to keep replicas *consistent*
    - Distribute replicas
    - Propagate modifications
  - Cost >> benefit if access-to-update is very low
Replication as Scaling Technique

What if there is an update?

- **Update all in an atomic way** (sync replication)
- To keep replicas consistent → conflicting operations are done in the same order everywhere
  - ✓ Read–write conflict: read and write operations act concurrently
  - ✓ Write–write conflict: two concurrent write operations
- making the cure worse than the disease!

Solution

- Loosen the consistency constraint so that hopefully global synchronization can be avoided
Consistency Models

- Data-Centric
  - Multiple writers may update the data store simultaneously

- Client-Centric
  - Lack of simultaneous updates.
Consistent Ordering of Operations

- How to reach a **global order of operations** applied to replicated data so we can provide a system-wide consistent view on data store?

- Comes from concurrent programming
  - Sequential consistency
  - Causal consistency

\[ W_i(x)a: \text{a write by process } P_i \text{ to data item } x \text{ with the value } a \]
Sequential Consistency (1)

The result of any execution is the same as if the (R/W) operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program – by Lamport

Behavior of two processes operating on the same data item. The horizontal axis is time.

\[
P1: \quad W(x)a \\
P2: \quad R(x)NIL \quad R(x)a
\]

it took sometime to propagate new value of x
Sequential Consistency (2)

(a) A sequentially consistent data store.

(b) A data store that is NOT sequentially consistent. Why?

- Any valid interleaving of R and W is acceptable as long as all processes see the same interleaving of operations.
- Everyone sees all W in the same order
Weakening sequential consistency
- NOT all, only causally related W → seen in same order

It implies:
- Writes that are potentially causally related must be seen by all processes in the same order.
- Concurrent writes may be seen in a different order on different machines.

If event b is caused by an earlier event a, a → b
- P1: Wx  P2: Rx then Wy, then Wx → Wy (potentially causally related)
Causal Consistency (2)

This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

(a) A violation of a causally-consistent store

(b) Causally but not sequentially consistent events.

Implementing causal consistency requires keeping track of which processes have seen which write → Construct a dependency graph using vector timestamps…
Client-Centric Consistency Models

- **Data-centric**: aiming at providing a systemwide consistent view on a data store.
  - Assumption: processes can update simultaneously the data store, thus it is necessary to provide consistency.
  - Sequential is good but costive, only guarantee when using transactions or locks.

- **Client-centric**: lacking of simultaneous updates, or we only care about when updates happen.
  - From a **specific client** point of view.
Eventual Consistency (1)

- Most processes hardly ever perform updates while a few do updates.
- How fast updates should be made available to only reading processes (e.g., DNS)
  - Consider WWW pages, not write-write conflict
    - To improve performance clients cache web pages. Caches might be inconsistent with original page for some time...
    - Eventually all will be brought up to date
  - MongoDB, CouchDB, Amazon DynamoDB and SimpleDB
- Eventual consistency:
  If no updates take place for a long time, all replicas will become consistent.
Pull versus Push Protocols

- Pushing updates:
  - server-initiated, in which update is propagated regardless whether target asked for it. + good if r/w is high: read more

- Pulling updates:
  - client-initiated: + good if r/w is low: write more, read less

- We can dynamically switch between pulling and pushing using leases (a hybrid form):

- Lease is a contract in which the server promises to push updates to clients until the lease expires.
Lecture 13: Fault Tolerance

- Terminology: fault, error and failures
- Fault recovery techniques
  - Redundancy: time and space
- Recovery and rollback
  - Checkpointing and stable storage
- Process resilience and reliable process groups
- Reliable communications
- Recovery in distributed systems:
  - Consistent checkpointing and message logging
Fault Tolerance Properties

- **Availability**
  - What percentage of time is a system available for use?

- **Reliability**
  - How long can a system run continuously without failure? Fail 1ms out of 1 our, availability > 99.9999% but not reliable

- **Safety**
  - Small failures should not have catastrophic effects

- **Maintainability**
  - How easy is it to repair faults?
  - High reliability = high availability!
Failure Models

- **Crash** failure
  - component simply halts, but behaves correctly before halting

- **Omission** failure
  - component fails to respond

- **Timing** failure
  - correct output, but lies outside a specified real-time interval

- **Response** failure
  - incorrect output

- **Arbitrary/Byzantine** failure:
  - Arbitrary/Malicious output

Crash failures are the least severe; arbitrary failures are the worst
Fault Management

- **Fault prevention**
  - prevent the occurrence of a fault

- **Fault tolerance**
  - build a component in such a way that it can meet its specifications in the presence of faults (i.e., mask the presence of faults)

- **Fault removal**
  - reduce the presence, number, seriousness of faults

- **Fault forecasting**
  - estimate the present number, future incidence, and the consequences of faults
Fault Tolerance Techniques

- **Redundancy** and agreement
  - Hiding effect of faults

- **Recovery** and rollback
  - Bringing system to a consistent state
Redundancy Techniques

- **Redundancy** is the key technique to tolerance faults

- **Information** redundancy
  - e.g., parity and Hamming codes

- **Time** redundancy
  - e.g., re-execution or execute secondary/backup copy

- **Physical** (software/hardware) redundancy
  - e.g., extra cpus, multi-versions softwares
Triple Modular Redundancy (TMR)

- If A2 fails → V1: majority vote → all B get good result
- What if V1 fails?!
TMR (cont.)

Correct results are obtain via **majority vote**

- Mask **ONE** fault

![Diagram](image)

(b)
Level of Redundancy

- Depends on
  - How many faults can a system handle?
  - What kind of faults can happen?

- $k$-fault tolerant system
  - Can handle $k$ faulty components

- Assume crash failure semantics (i.e., *fail-silent*)
  - $k + 1$ components are needed to survive $k$ failures
Level of Redundancy (cont.)

- Assume **arbitrary** (but **non-malicious**) failure semantics and group output defined by voting
  - Independent component failures $\rightarrow$ possible same results
  - $2k+1$ components are needed to survive $k$ component failures (**majority vote**)

- Assume **Byzantine (malicious)** failure semantics and group output defined by voting
  - Faulty components cooperate to cheat!!!
  - $3k+1$ components are needed to tolerate $k$ failures $\rightarrow$ two-thirds are needed for the agreement from other $2k+1$ non-faulty components;
Fault Recovery

- **Main idea**: when a failure occurs, we need to bring the system into an **error-free** state

- **Forward** recovery
  - Find a new state from which system continue operation
  - E.g., Error-correction codes
  - Problem: how to correct errors and move to a new state

- **Backward** recovery
  - Bring the system back into a **previous** error-free state
  - E.g., packet retransmission
  - Problem: keeping error-free state (checkpoints)
Recovery with Checkpoints

- Initial state
- Recovery line
- Checkpoint
- Message sent from P2 to P1
- Inconsistent collection of checkpoints
- Failure
Independent Checkpointing

- Each process independently takes checkpoints
  - Let $CP[i](m)$ denote the $m^{th}$ checkpoint of process $Pi$ and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$
  - When process $Pi$ sends a message in interval $INT[i](m)$, it piggybacks $(i,m)$
  - When process $Pj$ receives a message in interval $INT[j](n)$, it records the dependency $INT[i](m) \rightarrow INT[j](n)$
  - The dependency $INT[i](m) \rightarrow INT[j](n)$ is saved to stable storage when taking checkpoint $CP[j](n)$
- If process $Pi$ rolls back to $CP[i](m-1)$, $Pj$ must roll back to $CP[j](n-1)$. 

\[ \begin{array}{c}
P_i \\
CP[i](m-1) \quad INT[i](m) \quad CP[i](m) \\
\hline
\end{array} \]

\[ \begin{array}{c}
P_j \\
CP[j](n-1) \quad INT[j](n) \quad CP[j](n) \\
\hline
\end{array} \]

\[ INT(i)(m) \rightarrow INT[j](n) \]
Most important points

- Threads vs processes
- Multithreading performance
- CS and Synchronization Methods
- Virtual memory management
- Synchronization of distributed systems
- Consistency Model
Ph.D. Qualification Exams

- Open for master students and valid for four years. Petition through Dr. Zhang.
- One hour examination
- Analyze the performance and design solutions for given (maybe actual) problems
1. Virtual memory (VM) was designed traditionally to use small physical memory to support a larger address space.

(a) One important aspect of a modern paged virtual memory system is page size. What are the tradeoffs to have small page size and large page size?

(b) Page table structure is used to support virtualization. Very early systems used single-level page table. What are the advantages and disadvantages of using a single-level page table?