Concurrent Processes and Programming

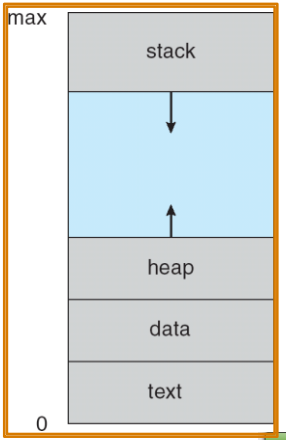
Topic 2

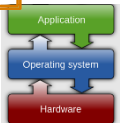
Hartmut Kaiser

https://teaching.hkaiser.org/fall2025/csc7103/

Processes

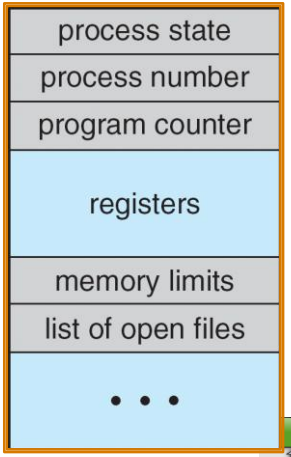
- Process (or task) – a program in execution:
  - Is an active entity – a program is passive entity until it is in execution
  - Is a unit of CPU work (all CPU activities are processes)
  - Executes OS and user codes
- A process consists of execution environment and one or more threads:
  - Address space
  - Synchronization, communication and scheduling information
  - Higher level resources like files
- A process includes:
  - Program counter and processor's registers defining current activity
  - Stack, heap, data and text
  - Process control block (PCB)





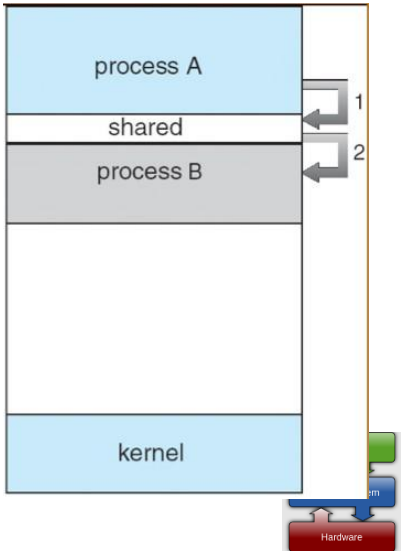
Process Control Block (PCB)

- Information associated with each process is represented by PCB
  - Process state: New, ready, running, waiting, and terminated
  - Process number: pid
  - Program counter: Indicates the address of next instruction to be executed for this process
  - CPU registers: Accumulators, registers, stack pointers
  - CPU scheduling/synchronization information: Process priority, scheduling queue pointers, semaphores, etc.
  - Memory-management information: Base and limit registers, page tables, etc.
  - Accounting information: CPU or real time used, time limits, account numbers, process numbers
  - I/O status information: I/O devices, files



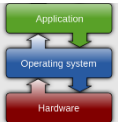
Operations on Processes

- Processes can execute concurrently, may be created and deleted dynamically
- Mechanisms for process creation and termination
  - Parent process can create child processes, which, in turn create other processes, forming a tree of processes
  - Creation of a new process:
    - `fork()` copies execution environment
    - `exec()` loads an executing program
- Process termination:
  - Normal (`exit`) or abnormal (`abort`), cascading termination
- Cooperating processes can share regions in their address space
  - Shared memory: Libraries, kernel code, share data in address space rather than communication by message passing
  - Inter Process Communication (IPC) can be done through shared memory



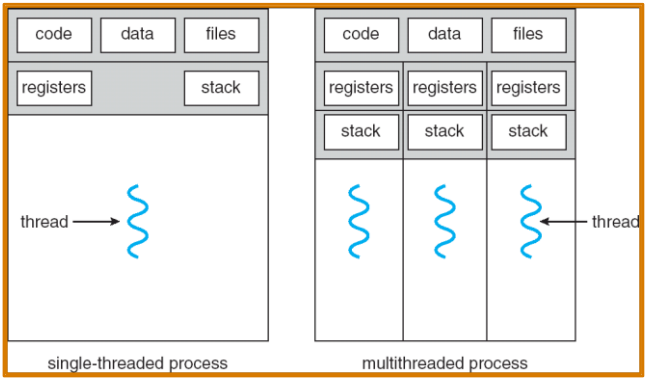
Threads

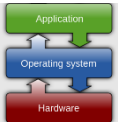
- Process can have
  - Single or multiple thread(s) of control or activity (strand of execution)
- A thread is a flow of control within a process
  - Basic computational unit – originally called a LWP (lightweight process)
  - Thread control block (TCB) consisting of thread ID, program counter, register set and stack
  - All threads share the same address space of their process
  - Thread support has become an integral part of any modern OS
- Multithreaded computer systems are common (e.g., desktop PCs)
  - Web browser: one thread for display and the other for data retrieving
- Benefits:
  - Economical (ease of creation, cheaper/quicker context switching)
  - Increased responsiveness
  - Simple and efficient resource sharing
  - Concurrency (multiprocessor architecture)



Single and Multithreaded Processes

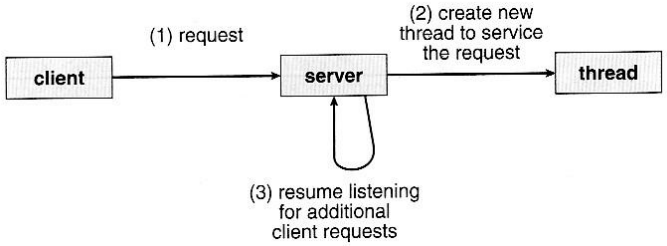
- Threads belonging to a given process share with each other code section, data section and other resources, e.g., open files

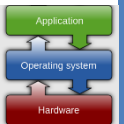




Multithreaded Server Architecture

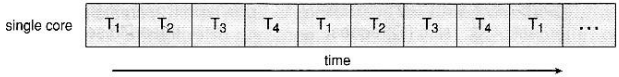
- For multi-threaded web server process, the server will create a separate thread that listens for client requests
- Using one process that contains multiple threads
- When a request is made, the server will create a new thread to service the request and resume listening for additional requests



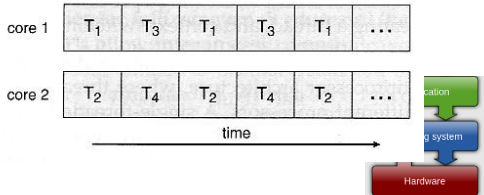


Multicore Programming

- A multi-threaded application running on a single-core system has to interleave the threads over time
  - Four threads T1 to T4 are time-interleaved

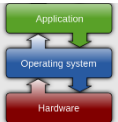


- Multiple cores (CPUs) on a single chip are common
- On a multi-core chip, we can spread the threads across the available cores to run them concurrently
  - Improved concurrency – threads can run in parallel



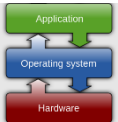
Kernel Space Thread Implementation

- Threads can be implemented at the kernel level
  - Blocking and scheduling of threads are treated normally by OS
  - Thread can be preempted easily
  - Thread issuing system call can be blocked without blocking other threads
  - Each thread competes for processor cycles on equal basis with processes
- Disadvantages
  - Two-level abstraction for concurrency becomes blurred
  - Context switching requires larger overhead
  - Portability becomes more difficult
- Examples: All operating systems support kernel threads - POSIX, Win32, Linux threads



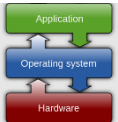
User Space Thread Implementation

- Thread support as an add-on package implemented on many systems
  - Threads run on top of a run-time library and visible to users
  - Thread library include thread primitives:
    - Thread management (creation, suspension, termination)
    - Assignment of priority (scheduling) and other thread attributes
    - Synchronization and communication support
- User threads are easy to create and manage, and are portable
  - The run-time procedure performs context switching from one thread to another, and handles blocking system call
  - Fast context switching as it involves saving and restoring only the program counter and stack pointers
  - Users have the option of setting thread scheduling criteria
  - Allocated processor time for a process is multiplexed among its all threads
- Examples: DCE thread package, Sun LWP package, POSIX



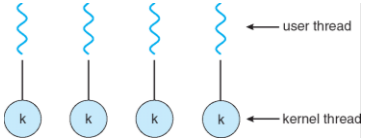
Combined (Hybrid) Implementation

- A combined user and kernel space implementation of threads captures the advantages of both approaches
- Single-threaded versus multiple-thread kernel
  - Operations internal to the kernel and all kernel services for user space applications can be implemented as threads
- Advantages
  - Support concurrent kernel services
  - Kernel space threads are multiplexed on the underlying multiprocessor system
- Thread executions are truly parallel and can be preemptive
  - Synchronization among kernel threads is necessary and can be done through shared memory

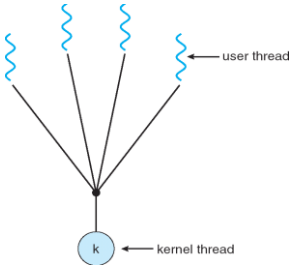


Multithreading Models

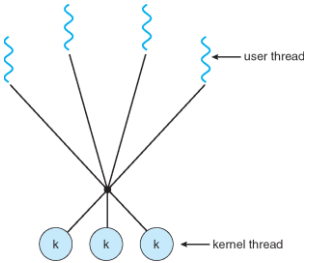
- Three common ways of establishing a relationship between user-level threads and kernel-level threads



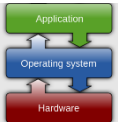
One-to-One



Many-to-One

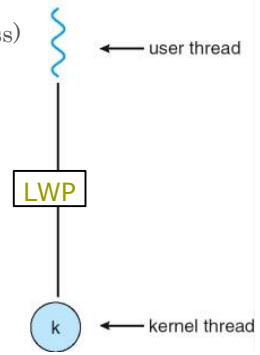


Many-to-Many



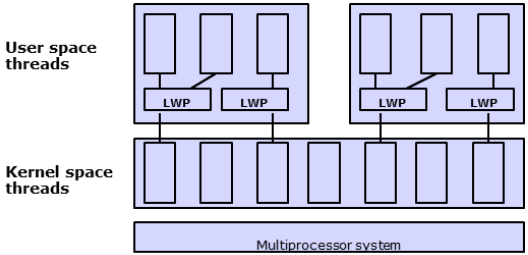
Interface Between User and Kernel Threads

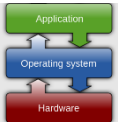
- Communication between the user-thread library and the kernel threads – scheduler activation
- An intermediate data structure known as LWP (light- weight process)
  - LWPs serve as a bridge between user thread and kernel thread
  - User thread runs on a virtual processor (LWP)
  - Each LWP is connected to a kernel thread
- Corresponding kernel thread runs on a physical processor
  - Each application gets a set of virtual processors (LWPs) from OS
- Application schedules (user-level) threads on these processors
  - When attached to a LWP, thread becomes executable
- Kernel informs an application about certain events issuing upcalls, handled by thread library



Multithreaded Kernel

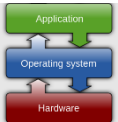
- Most OS kernels are now multithreaded
  - Each thread performing a specific task such as managing memory, interrupt handling, managing devices
- A preemptive multithreaded kernel with three levels of concurrency
  - User threads are multiplexed on LWPs in same process. LWPs are multiplexed on kernel threads, and kernel threads are multiplexed on multiple processors





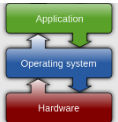
Concurrent Processes

- A process is sequential if a single thread of control regulates its execution
- Concurrent processes are simultaneous interacting sequential processes
  - They are asynchronous and each has its own logical address space
  - Disjoint components can be executed concurrently whereas others need to communicate and/or have synchronization between the processes
  - A concurrent system supports more than one task (process) by allowing all tasks to make progress unlike a parallel system where tasks actually run simultaneously
- Allowing multiple threads of control in a process introduces a new level of concurrency in the system
- A process may create new processes, thus creating multiple threads of execution



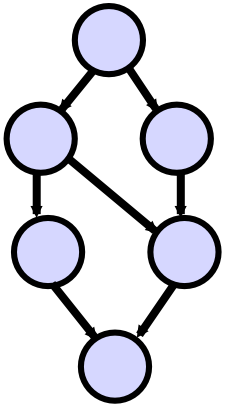
Process Representation

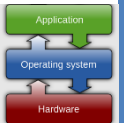
- To analyze the performance requirements, the processes should be represented in abstract ways, hiding unnecessary details
  - Processes are related by their need for synchronization and/or communication – cooperating processes
- Different models for process representation:
  - Graph models:
    - Synchronous process graph
    - Asynchronous process graph
- Process graph models are suitable for performance analysis
  - Space-time model: Useful for describing the detail interaction among the processes
- Language constructs or OS supports for concurrent processing:
  - cobegin/coend control structure
  - fork/join system calls



Synchronous Process Graph

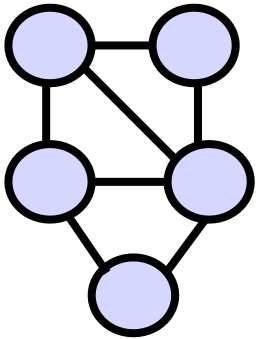
- Synchronization: Execution of some processes must be serialized in certain order
- Synchronous process graph - Direct Acyclic Graph (DAG)
  - Shows explicit precedence relationships and a partial ordering of a set of processes
  - Can be used to analyze the make-span (total completion time)
- Directed edges represent a synchronous communication of a sent or received message
  - Output results from a process are passed to a successor process as input
  - Communication transaction happens and is synchronized only at the completion of a process and the beginning of its successor process(es)

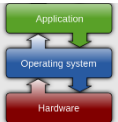




Asynchronous Process Graph

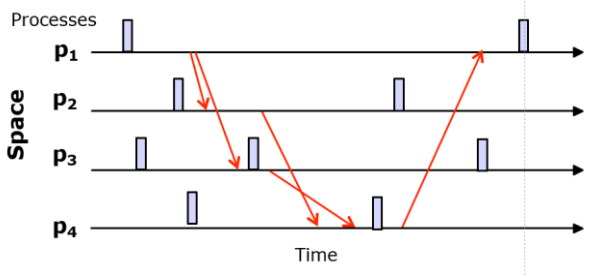
- Asynchronous process graph represents communicating processes without any reference to the precedence relation
  - Undirected edges indicate the existence of communication paths but say nothing about how and when communication occurs
  - Can be used to study processor allocation for optimizing inter-processor communication overhead
- Three types of communication scenarios:
  - One-way: simplex communication
  - Client/server: half-duplex communication
  - Peer to peer: full-duplex communication

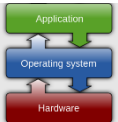




Space-Time Model

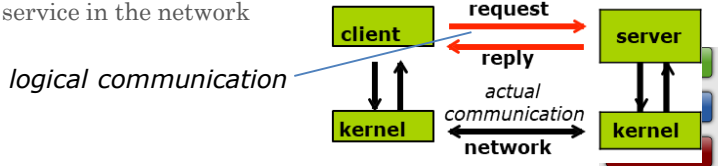
- The events along with their time of occurrence are shown – timing diagram
- Provides greater details: actual communications are explicit
  - Useful to analyze the instantaneous interactions among processes
  - Both synchronous and asynchronous process graphs can be derived





The Client/Server Model

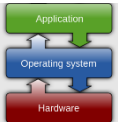
- The client/server model is a programming paradigm that represents the interaction between processes and structures of the system
- Any process in a system is a client or a server or can play the roles of both
  - Client and server processes interact through a sequence of requests and responses: synchronous request/reply exchange of information
- Can also be considered as a service-oriented communication model
  - A higher-level abstraction of inter-process communication
  - Supported by using either RPC or message passing communication, which in turn is implemented (lower level) by either a connection-oriented or connectionless transport service in the network



Time Services

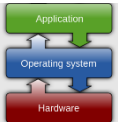
Time Services

- In the space-time model, events are recorded with respect to each process's own clock time
- Many applications need timing information: clocks are used to specify time and timers needed for describing the occurrences of events:
  - When an event occurs, how long it takes, and which event occurs first
  - Computer example: When a file was last modified, how long a client should access a server, and which update of a data object happened first
  - Two processes updating a file: The file server received two simultaneous write requests. Without a time stamp, it is not possible for the server to determine which request is older
- It is not possible to precisely synchronize clocks in a distributed system because different computers have their own (local) clocks
  - Even if the clocks may be synchronized, physical clocks will not run at the same pace so clock skew (clock drift) will develop over time
- To ensure that different computer clocks in reasonable agreement need clock synchronization protocols



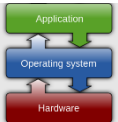
Two Types of Clocks

- In centralized OS, the same system clock and time/timer are used to coordinate all processes/events
- In distributed OS, without a global time consensus, it is difficult to coordinate distributed processes/events
- Two fundamental clock concepts for specifying time in distributed systems
  - Physical clock: a close approximation of real life clock that measures both a point and intervals of time, i.e., time and timer
    - Used to synchronize and schedule hardware activities
  - Logical clock: preserves only the ordering of events
- Different algorithms to synchronize physical and logical locks



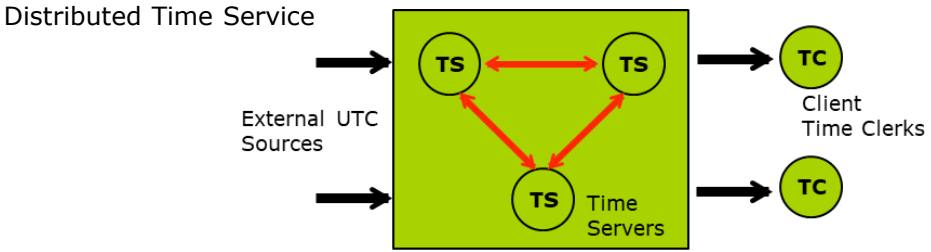
Physical Clocks

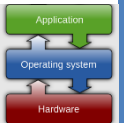
- An absolute global physical time is theoretically impossible to obtain in a distributed environment
  - All machines try to reach a consensus of time
- Many standard Universal Coordinated Time (UTC) sources are available for computers:
  - NIST short wave radio, ACTS (via modem), GPS satellites
  - Using these time services are generally costly and/or complicated. A few time services access the UTC sources directly, and most computers obtain the timing information from one or more time servers (TS)
- Accessing a TS via network requires a non-deterministic message exchange time – a delay in reporting or acquiring UTC
  - Approaches to compensate the delay and reduce time discrepancies: Christian's Algorithm
  - Berkeley Algorithm
  - Averaging Algorithm



A Distributed Time Service Architecture

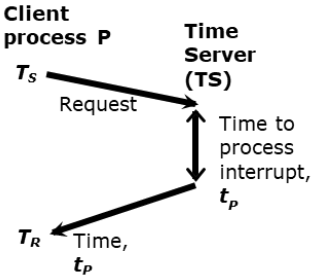
- Time clerk on client machine requests time service from time server
  - Time servers maintain up-to-date clock information and also exchange the timing information
- Two key issues: compensating delay and calibrating discrepancy
- Two ways of accessing UTC: pull and push service models

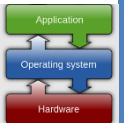




Christian's Algorithm

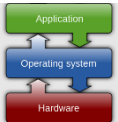
- A client process P requests a time service from a time server (TS) at time T_S
- The time server responds with time T_C from its clock, which is received by the client at T_R
- However, the UTC (T_C) returned from the time server to the client must be adjusted:
  - If $\mathbf{d}$ is the estimate of delay from the server to the client, the client should then set its clock/time to $(\mathbf{T_C + d})$
  - If nothing is known about $\mathbf{t_P}$, then estimated $\mathbf{d = (T_R - T_S)/2}$ ($\mathbf{t_p}$ is assumed to be zero)
  - If $\mathbf{t_P}$ is known, then estimate $\mathbf{d = (T_R - T_S - t_P)/2}$
  - Or $\mathbf{d}$ may be estimated using average or minimum of multiple requests





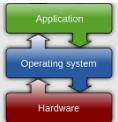
Berkeley Algorithm

- This algorithm does not use the actual time for the time server (TS) but rather maintains a global time
- The time server periodically gathers time data from all other computers (clients) in the distributed system and determines the average time. Then the server communicates this global average time (in fact the difference) to all machines
  - All clients adjust their local clocks accordingly
- Suitable for systems which can not access any UTC server



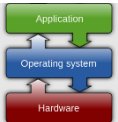
Averaging Algorithm

- The Berkeley algorithm is centralized and has single point of failure
- Every computer broadcasts its time information at a regular interval known system wide
  - Every machine then averages the broadcast information to compute its own time and reset its internal clock accordingly
- Both the Berkeley and averaging algorithms considers the fact that local clocks not only contain different times but they also have rates.
  - Any times (or UTCs) that are suspiciously small or large (differing by a value outside of a given tolerance) are excluded from the computation
  - This prevents the overall system time from being drastically skewed due to one or more erroneous clocks
- Timing discrepancies can be significantly reduced but a small degree of uncertainty is inherent in UTC or global time: $t_p \pm \Delta l$



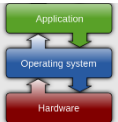
Clock Adjustment

- Note that the clock of the system can not be set backwards. Doing so may result in an inconsistent system state
- To solve the problem the clock is slowed down for a certain period of time until the system clock reaches the desired state
  - If the local time in the client is higher than the new UTC, its clock speed is slowed by the software
- Problem with a slower clock is less serious
  - Increase clock speed to allow it catch up the UTC gradually
- How often to synchronize:
  - It is possible to put an upper bound (Δ) on the rate at which the clock skew develops. That means, the clock in a system may deviate at most Δt from the real clock over time period t
  - Therefore, if two clocks are synchronized at time 0, then they will differ by at most $2\Delta t$ at the end of time period t (one slower, one faster)



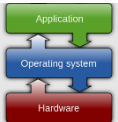
Logical Clocks

- Physical clocks can generally tell whether an event happens before another
  - Problem arises when two events occur very closely
  - The uncertainty of UTC is high or UTC intervals of events overlap with each other
  - This can be the case in distributed systems
- Logical clocks can be used to indicate the ordering information for events
  - The ordering of event execution can be determined without scheduling/ synchronizing the events with respect to the real-time clock
- Three levels of logical clocks can be used:
  - Lamport's logical clocks
  - Vector logical clocks
  - Matrix logical clocks



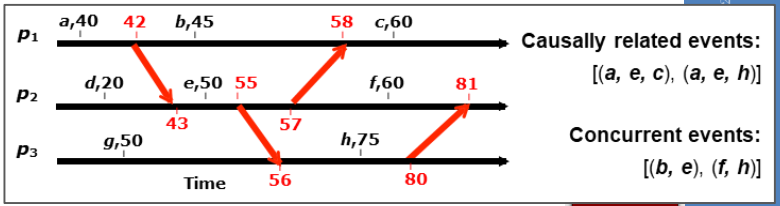
Lamport's Logical Clock

- Lamport's logical clock is for ordering processes and events:
  - Each process (processor) p_i in the system maintains a logical clock $C_i(T)$ and the time is monotonically non-decreasing
    - The logical clock is always incremented by an arbitrary positive number when events in the process progress
- The algebraic notation $\rightarrow$ defined as the happens-before relation is used to synchronize the logical clocks between two events
  - $a \rightarrow b$ means event a precedes event b: If events are within the same process, then logical clocks satisfy $C(a) < C(b)$
  - Processes interact with each other through a pair of sends/receives
  - Logical clocks for send event (by process p_i) and receive events (by process p_j) must satisfy C_i (send) < C_j (receive)
  - The happens-before relation describes the causality between two events
  - Two events are disjoint and can be run concurrently
    - If neither $a \rightarrow b$ nor $b \rightarrow a$ is true, logical clocks not relate to each other



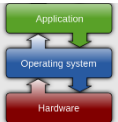
Lamport's Algorithm

- Lamport's algorithm for logical clocks:
  - Rule 1: The events within a processor (same process) may be consistently ordered using the system clock: If **a** → **b** then **C(a) < C(b)**
  - Rule 2: The ordering of the events occurring in different processors (processes) do not matter as long as it is consistent with the communication pattern
    - If **a** is the sending event of p_i and **b** is the corresponding receiving event of p_j, then $C_i(a) < C_j(b)$
- The sending process (p_i) timestamps its logical clock time, $C_i(a)$, in the message and the receiving process (p_j) updates its logical clock as
- $C_j(b) = \max(C_i(a)+d, C_j(b))$ where d = 1



Partial and Total Ordering of Events

- Note that the described version of Lamport's algorithm provides only a partial ordering of the events across the system
  - For two disjoint events **a** and **b**, $C_i(a) < C_j(b)$ does not imply $a \rightarrow b$
- If a total ordering of the events is necessary, the following additional rule can be imposed: For all events **a** and **b**, $C(a) \neq C(b)$
  - To distinguish disjoint events with identical logical time clocks, their time stamps (logical clock times) can be appended with the processor/ process **id**
  - Ensure system-wide unique logical clock times for all events

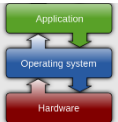


Vector Logical Clocks

- Problem with Lamport's algorithm: Does not help to recognize the concurrent events as two concurrent events may have different times stamps
  - If $C_i(a) < C_j(b)$, it is not possible to tell whether event **a** causally (actually) happened before event **b**, or whether they are concurrent
  - The problem can be solved using vector logical clock
- Each process p_i maintains an array (vector) of logical clocks for each event:

$$VC_i(a) = [TS_1, TS_2, \ldots\ldots\ldots, C_i(a), \ldots., TS_n]$$

- where the i^{th} entry corresponds to the logical clock (TS_i) for event **a** at processor **i**, and TS_k **(k = 1, 2…. n, except i)** is the best estimate of the logical clock time for process p_k



Vector Logical Clock Ordering

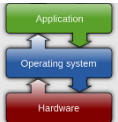
- If all coordinates of $VC_i(a)$ are less than or equal to the corresponding coordinates of $VC_j(b)$ and at least one coordinate of $VC_i(a)$ is strictly less than that of $VC_j(b)$, then we have

$$VC_i(a) < VC_j(b)$$

  - This means that event **a** in p_i happened before event **b** in p_j

- The vector logical clock is maintained in such a way that the time stamp for two concurrent events will be incomparable
  - If neither $VC_i(a) < VC_j(b)$ nor $VC_j(a) < VC_i(b)$, then they are incomparable

- If the time stamps are comparable, then the event with smaller time stamp occurred before



Vector Logical Clock Update

- Process $\mathbf{i}$ maintains its vector logical clock as follows:

  VC1: Initially, $\mathbf{VC_i = [0....0]}$ for each process $\mathbf{i}$

  VC2: Just before process $\mathbf{p_i}$ timestamps an event $\mathbf{a}$ (i.e., an event of sending message $\mathbf{m}$ to process $\mathbf{p_j}$), its logical clock is incremented

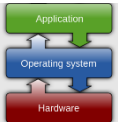
  VC3: $\mathbf{p_i}$ includes its logical timestamp $\mathbf{VC_i(m)}$ in message $\mathbf{m}$ to process $\mathbf{p_j}$

  VC4: When $\mathbf{p_j}$ receives a timestamp in message $\mathbf{m}$ (i.e., receive event $\mathbf{b}$), it updates its logical clock vector $\mathbf{VC_j(b)}$ such that

  $$\mathbf{TS_k(b) = max(TS_k(a), TS_k(b))}$$

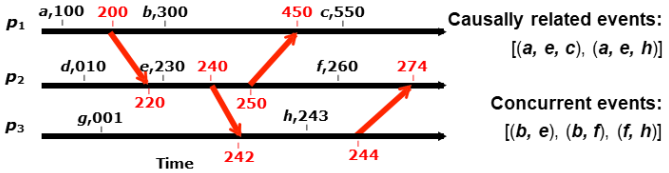
  That is, $\mathbf{p_j}$ takes the pair-wise maximum of the entries for every $\mathbf{k} = 1....\mathbf{n}$, and also increments its logical clock

- The most recent logical clock information is thus propagated to every process by sending timestamps $\mathbf{TS_k}$ in the messages



Vector Logical Clock Example

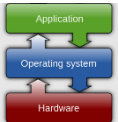
- Logical vector clock in space-time diagram is updated every time a process progresses with an event
  - When processor executes an event, it assigns the timestamp to the event
  - Processor attaches its timestamp to all messages that it sends



$VC_1(a) = [100,\ldots,\ldots]$, $VC_1(b) = [300,\ldots,\ldots]$
$VC_2(e) = [\ldots,230,\ldots]$, $VC_2(f) = [\ldots,260,\ldots]$
$VC_3(h) = [\ldots,\ldots 243]$

$VC_1(a) < VC_2(e) < VC_3(h)$ so events (***a, e, h***) are causally related

Neither $VC_1(b) < VC_2(f)$ nor $VC_2(f) < VC_1(b)$ holds for disjoint objects (***b, f***)



Two Timestamp Algorithms

Lamport's Timestamp Algorithm

Initially,

```
my_TS = 0
```

On event **e**,

```
if e is the receipt of message m
    my_TS = max(m.TS, my_TS)
my_TS[self]++
e.TS = my_TS
if e is the sending of message m,
    m.TS = my_TS
```

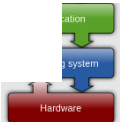
Vector Timestamp Algorithm

Initially,

```
my_VT = [0,…..0]
```

On event **e**,

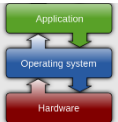
```
if e is the receipt of message m
    for i = 1 to M
        my_VT = max(m.VT[i], my_VT[i])
my_VT[self]++
e.VT = my_VT
if e is the sending of message m,
    m.VT = my_VT
```



Matrix Logical Clock

- The concept of vector logical clock can be extended to matrix logical clock
- Instead of an array of logical clocks, an array (vector) of vector logical clocks is maintained
  - A matrix clock at process P_i is an n by n matrix:

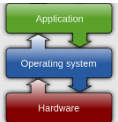
    $MC_i [k, i]$
  - where i^{th} row $MC_i [i, 1...n]$ is the vector logical clock of p_i
- The j^{th} row is the knowledge process p_i has about the vector logical clock of process p_j
- The matrix thus obtained is used to stamp messages
  - Logical clock within a process is incremented for each local event
  - Upon receipt of a message, the matrix clock is updated by taking the pair-wise maximum



Causality

Causality

- The lack of a global system state – fundamental property of a distributed system
  - Most of the time distributed systems are asynchronous
- Distributed systems are causal – the cause precedes the effect
  - The sending of a message precedes the receipt of a message
  - The distributed system is composed of the set of processors and there are multiple sets of events that occur on these processors
    - Events include message send, message receipt, user input receipt, signal raising, output creation, etc.
  - How to define the ordering among different events:
    - We write $e_1 < e_2$ if we know that event e_1 occurred before event e_2
- In distributed systems, it is difficult to deduce which event came first
  - Need to combine information from different sources to determine the ordering. If information source I tells us that e_1 occurred before e_2, we write $e_1 <_I e_2$



Causality Definitions

- Event e_1 causally happened before event e_2 (that is, $e_1 <_H e_2$):
  - Transitive closure of the processor orderings and the message orderings
- Processor ordering: e_1 occurred before e_2 in the same process/processor p

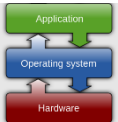
$$e_1 <_P e_2$$

  - Events that occur on the same processor are totally ordered
- Message ordering: A message (m) sent by the process p_i after e_1 occurred is received by the process p_j before e_2 occurred

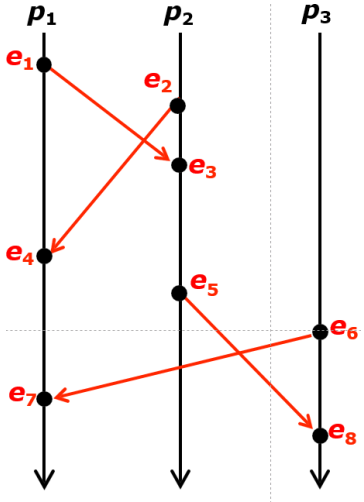
$$e_1 <_m e_2$$

  - Simply, e_1 is the sending of message m and e_2 is the receipt of message m
- Transitive closure property: if e_1 causally happened before e_2 and e_2 causally happened before e_3, then

$$e1 <_C e3$$



Happens-Before DAG



Causally ordered events:

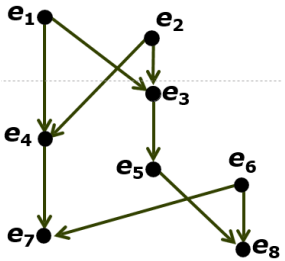
$e_1 <_{P1} e_4 <_{P1} e_7$

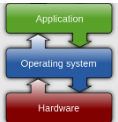
$e_1 <_m e_3$

Concurrent (disjoint) events

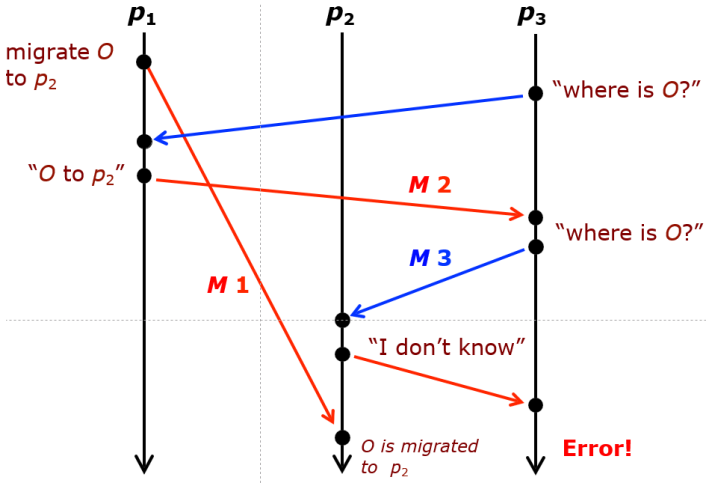
e_1 and e_6

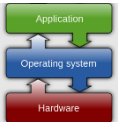
DAG (directed acyclic graph)





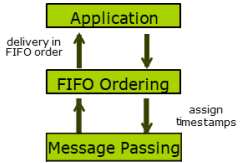
Causality Violation

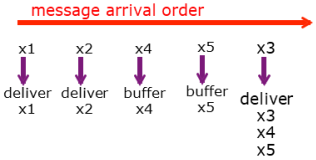


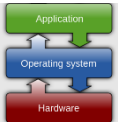


Causality Communication

- Ensuring that processor never experiences a causal violation
- Protocol for causal communication:
  - A processor cannot choose the order in which messages arrive but it can change the order in which messages are delivered to the applications that consume them
  - Revise delivery order by holding back messages that arrived "too soon"
  - The source attaches timestamps on messages (to order messages), and the destination delays the delivery of out-of-order messages
- Protocol for FIFO message delivery (TCP communication)



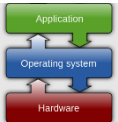




IPC and Synchronization

Language Mechanisms for Synchronization

- A concurrent programming language supports:
  - Specification of concurrent processing
  - Synchronization of processes
  - Interprocess communication
  - Non-deterministic execution of processes
- How the normal OS approaches can be extended to the distributed OS
- Various synchronization mechanisms
  - Shared-variable approaches: Semaphore, monitor, conditional critical region, serializer, path expression
  - Message passing approaches: Communicating sequential processes, remote procedure call, rendezvous
- Classic synchronization example: Concurrent readers/exclusive writer problem

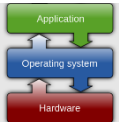


Critical Section Problem

- Multiple processes are competing to use some shared data
- Each process has a code segment, called critical section, in which the shared data is accessed
- Problem – ensure that when one process is executing in its critical section, no other processes are executing in their critical sections
  - Mutual exclusion should be enforced
- Entry section implements a process' request to enter its critical section which is followed by an exit section
  - Processes may share some common variables to synchronize their actions (to have orderly execution of cooperating processes)

General structure of process p_i

```
do {
    entry section
      critical section
    exit section
      reminder section
} while (1);
```



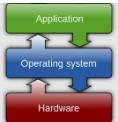
Semaphores

- Semaphore is a synchronization tool
  - Works like mutex locks to enforce mutual exclusion
- Semaphore S – protected integer variable which can only be accessed via two operations

```
wait(S){
    while (S≤ 0)
        ; // no operation (busy wait)
        S--;
}
signal (S){
    S++;
}
```

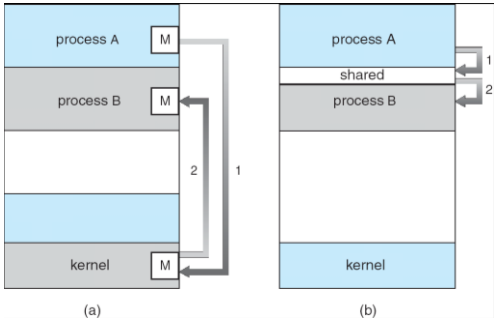
Original terminology:
P (from Dutch proberen, to test) for **wait**
V (from verhogen, to increment) for **signal**

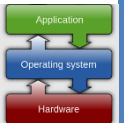
- These operations are indivisible (atomic), that is, only one process can modify the semaphore value at a time



Interprocess Communication Models

- Message passing – Useful for exchanging smaller amounts of data; easier to implement through system calls but slower
- Shared memory – Allows maximum speed and convenience of communication; faster accesses to shared memory





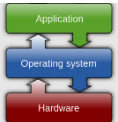
Message-Passing Synchronization

- A mechanism for cooperating processes to communicate and to synchronize their actions without sharing the same address space
  - The only means of communication in distributed systems without shared memory
- Message-passing facility provides two operations:

  `send(message)` – message size fixed or variable

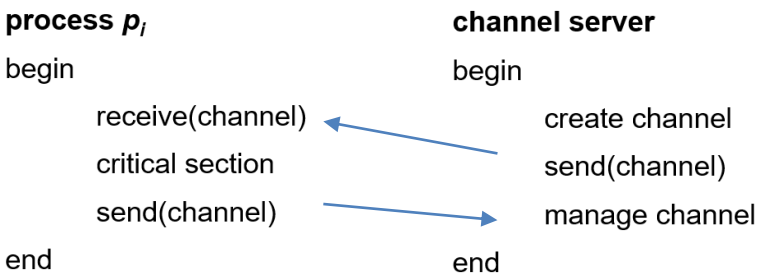
  `receive(message)`
- Two processes wishing to communicate need to establish a communication link between them and exchange messages via

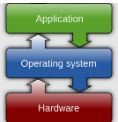
  `send/receive`
  - Implementation of communication link physical (e.g., shared memory, hardware bus or network) or logical
- Message can be asynchronous or synchronous



Asynchronous Message Passing

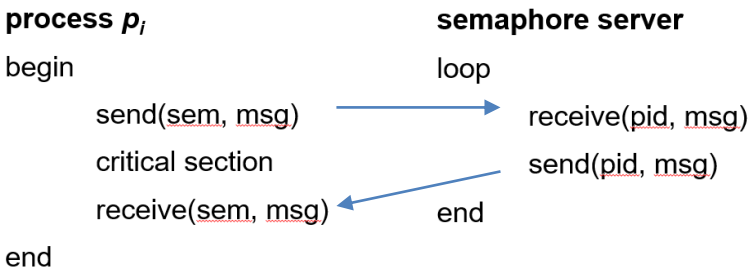
- Assumes non-blocking send and blocking receive – uses the channel with an unbounded buffer as a semaphore (message content is not important)
- Can be useful as semaphore if communication channel can be specified
  - Blocking receive – (acquiring the lock), Non-blocking send – (releasing the lock)
- Mutual exclusion solution using asynchronous message passing:

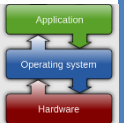


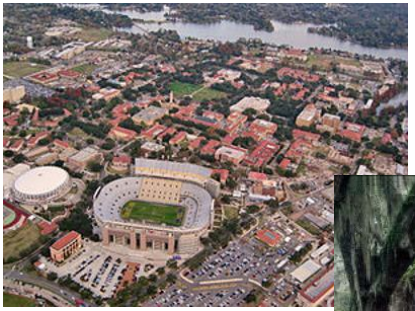


Synchronous Message Passing

- Assumes blocking send and blocking receive – symmetrical waiting
- Rendezvous between send and receive
  - Allows two processes to join and exchange data at a synchronization point and continue their separate execution thereafter
- Mutual exclusion solution using synchronous message passing:







LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP