SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 431 Distributed Systems

Fall 2019

Distributed Computing

https://www.cs.sfu.ca/~keval/teaching/cmpt431/fall19/

Instructor: Keval Vora

Parallel Computing

- Topics discussed so far assumed a single machine
- Logic was distributed, but direct co-ordination was possible
  - E.g., signaling via variables, helper threads doing insertion

- Distributed: each processes on a different machine
- Processes/threads have their own memory, clock, etc.
- Coordination happens via communication

Reading

- [DC] Chapter 1 (1.1, 1.6-1.8)

Distributed System

- Autonomous processors (or nodes) communicating over a communication network

- No common physical clock
- No shared memory
  - Communication via message passing
- Autonomy and heterogeneity
  - Nodes have different speeds, operating systems, etc.
- Geographical separation
  - LAN, geo-distributed, or even VMs on same physical machine

Parallel Sum v/s Distributed Sum

```
int sum = 0

void accumulate(int x) {
  int old, new;
  do {
     old = sum;
     new = old + x;
  } while(!cas(&sum, old, new));
}
```

```
int sum = 0

void accumulate(int x) {
    if(pid != p0) {
        send(p0, x);
        sum = receive(p0);
    }
    else {
        for p in peers {
            sum += receive(p);
        }
        sum += x;
        for p in peers {
            send(p, sum);
        }
    }
}
```

Communication Primitives

- Synchronous (send/receive)
  - Handshake between sender and receiver
  - Send completes when matching receive completes
  - Receive completes when data copied into buffer
- Asynchronous (send)
  - Control returns to process when data copied out of user-specified buffer

Communication Primitives

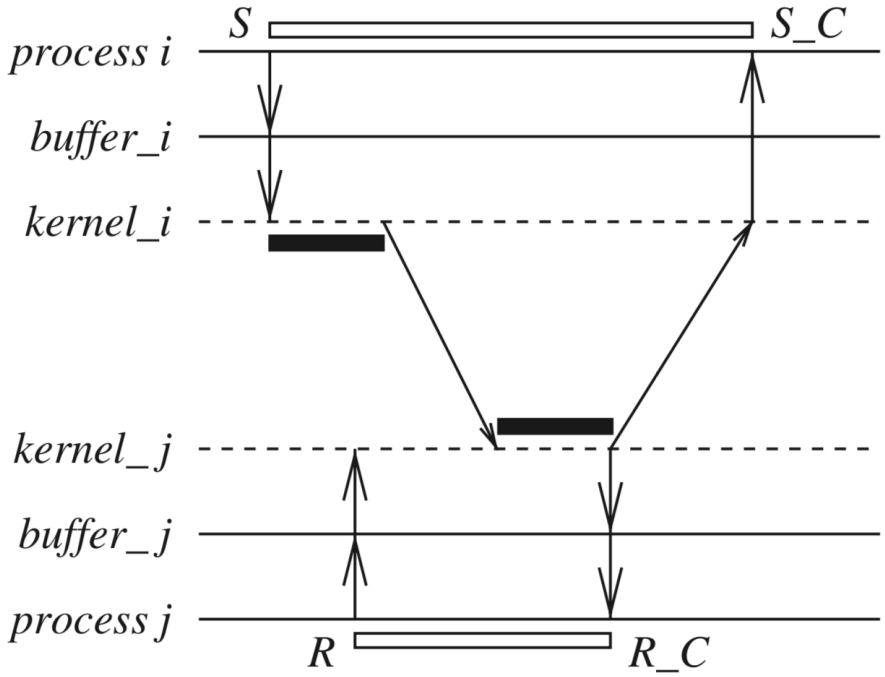
- Blocking (send/receive)
  - Control returns to invoking process after processing of primitive (whether sync or async) completes

- Non-blocking (send/receive)
  - Control returns to process immediately after invocation
  - Send: even before data copied out of user buffer
  - Receive: even before data may have arrived from sender

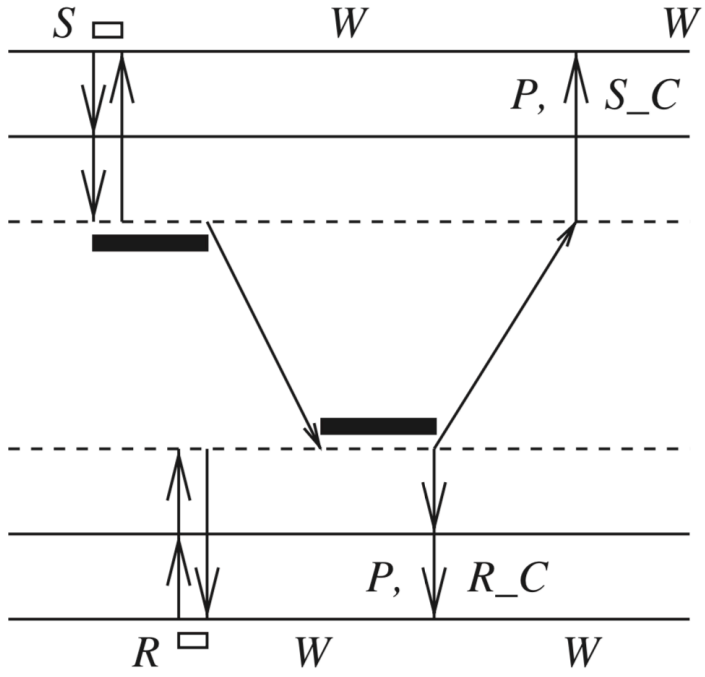
Non-blocking Primitives

- Return parameter returns a system-generated handle
  - Used to check for status of call
  - Periodically keep checking if handle has been posted
  - Issue `Wait(handle1, handle2, . . .)` with list of handles
  - Wait call blocks until one of the stipulated handles is posted

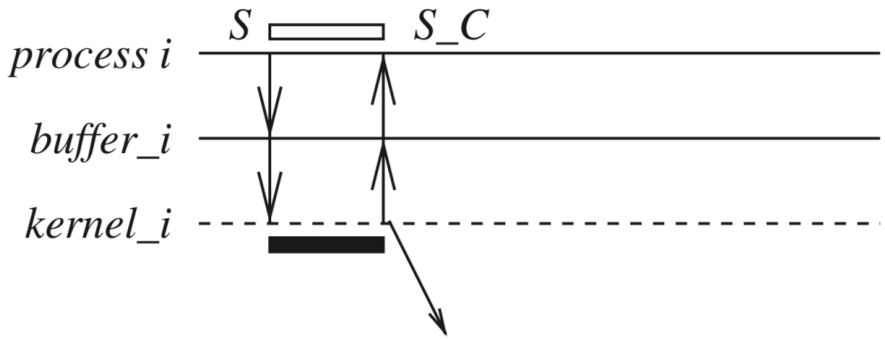
```
handle = send(value, destination)
do_some_work();
wait(handle)          // blocking
```



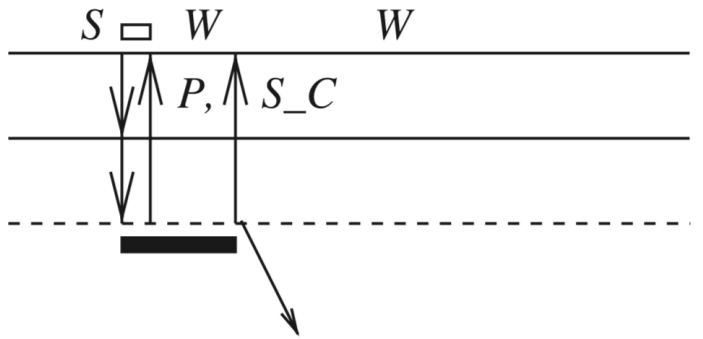
(a) Blocking sync. *Send*, blocking *Receive*



(b) Nonblocking sync. *Send*, nonblocking *Receive*



(c) Blocking async. *Send*



(d) Non-blocking async. *Send*

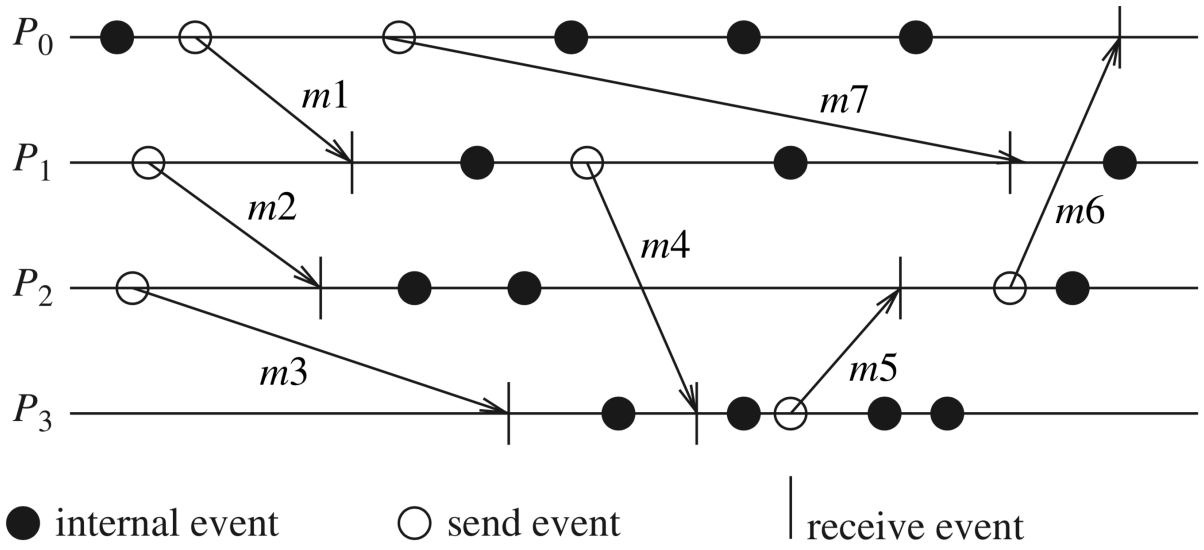
Communication Primitives

- Synchronous blocking send-receive
  - Easy to use since operations appear to happen at those points
  - Performance impact: send has to wait for matching receive which might not even have been posted

- Asynchronous/Non-blocking primitives
  - Allow hiding communication latencies and delays caused due to waiting (e.g., send waiting for receive to be posted)
  - Difficult to use
    - Need to reason about outstanding calls happening simultaneously
    - Need to verify the handle before proceeding

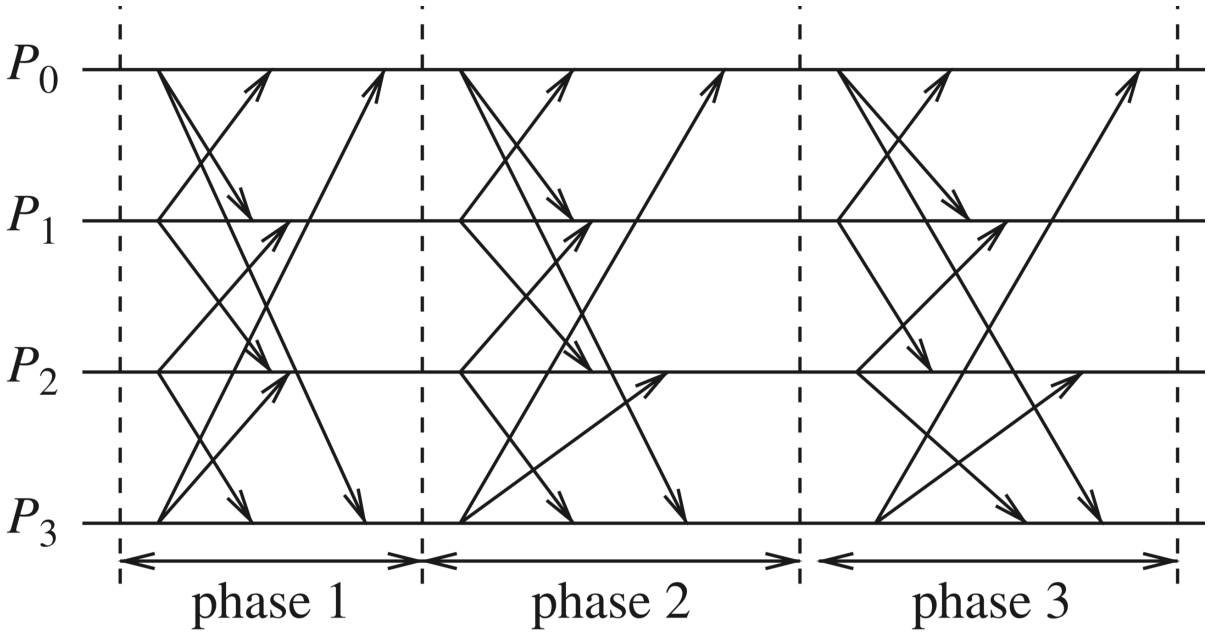
Asynchronous Execution

- No processor synchrony; No bound on drift rate of clocks
- Message delays are finite but unbounded
- No bound on time for a step at a process



Synchronous Execution

- Processors are synchronized; Clock drift rate bounded
- Message delivery occurs in one logical step/round
- Known upper bound on time to execute a step at a process



Sync v/s Async Executions

- Programming is easier in synchronous execution/system
- Building truly synchronous systems is challenging

- Synchrony is usually simulated by blocking or delaying certain operations
  - To provide an abstraction of synchronous execution
  - Depending on application needs
  - Fewer synchronization "steps" ➔ lesser delays

Distributed Systems Challenges (1)

- Communication mechanisms
  - Remote Procedure Call (RPC), remote object invocation (ROI), message-oriented vs. stream-oriented communication
- Processes
  - Code migration, process/thread management at clients/servers
- Naming
  - Easy to use identifiers needed to locate resources and processes in transparent and scalable manner
- Synchronization
  - Mutual exclusion, logical clocks, global state recording

Distributed Systems Challenges (2)

- Data storage and access
  - Schemes for data storage, search, and lookup should be fast and scalable across network; Revisit file system design
- Consistency and replication
  - Replication for fast access, scalability, avoid bottlenecks
  - Require consistency management among replicas
- Fault-tolerance
  - Correct and efficient operation despite link, node, process failures

Distributed Systems Challenges (3)

- Distributed systems security
  - Secure channels, access control, key management (key generation and key distribution), authorization, secure group management
- API for communications, services
  - Ease of use
- Scalability and modularity of algorithms, data, services
  - Decentralized logic, caching

Distributed Systems Challenges (4)

- Transparency: hiding implementation policies from user
  - Access: hide differences in data representation across systems, provide uniform operations to access resources
  - Location: locations of resources are transparent
  - Migration: relocate resources without renaming
  - Relocation: relocate resources as they are being accessed
  - Replication: hide replication from the users
  - Concurrency: mask the use of shared resources
  - Failure: reliable and fault-tolerant operation