ECE/CS 757: Advanced Computer Architecture II

Instructor:Mikko H Lipasti

Spring 2017

University of Wisconsin-Madison

Lecture notes based on slides created by John Shen, Mark Hill, David Wood, Guri Sohi, Jim Smith, Natalie Enright Jerger, Michel Dubois, Murali Annavaram, Per Stenström and probably others

Lecture Outline

- Introduction to Parallel Software
  - Sources of parallelism
  - Expressing parallelism
- Programming Models
- Major Abstractions
  - Processes & threads
  - Communication
  - Synchronization
- Shared Memory
  - API description
  - Implementation at ABI, ISA levels
  - ISA support
- Message Passing
  - API description
  - Implementation at ABI, ISA levels
  - ISA support

Parallel Software

- Why is it so hard?
  - Conscious mind is inherently sequential
  - (sub-conscious mind is extremely parallel)
- Identifying parallelism in the problem
- Expressing parallelism to the hardware
- Effectively utilizing parallel hardware
  - Balancing work
  - Coordinating work
- Debugging parallel algorithms

Finding Parallelism

1. Functional parallelism
   - Car: {engine, brakes, entertain, nav, …}
   - Game: {physics, logic, UI, render, …}
   - Signal processing: {transform, filter, scaling, …}
2. Automatic extraction
   - Decompose serial programs
3. Data parallelism
   - Vector, matrix, db table, pixels, …
4. Request parallelism
   - Web, shared database, telephony, …

1. Functional Parallelism

1. Functional parallelism
   - Car: {engine, brakes, entertain, nav, …}
   - Game: {physics, logic, UI, render, …}
   - Signal processing: {transform, filter, scaling, …}

- Relatively easy to identify and utilize
- Provides small-scale parallelism
  - 3x-10x
- Balancing stages/functions is difficult

2. Automatic Extraction

2. Automatic extraction
   - Decompose serial programs

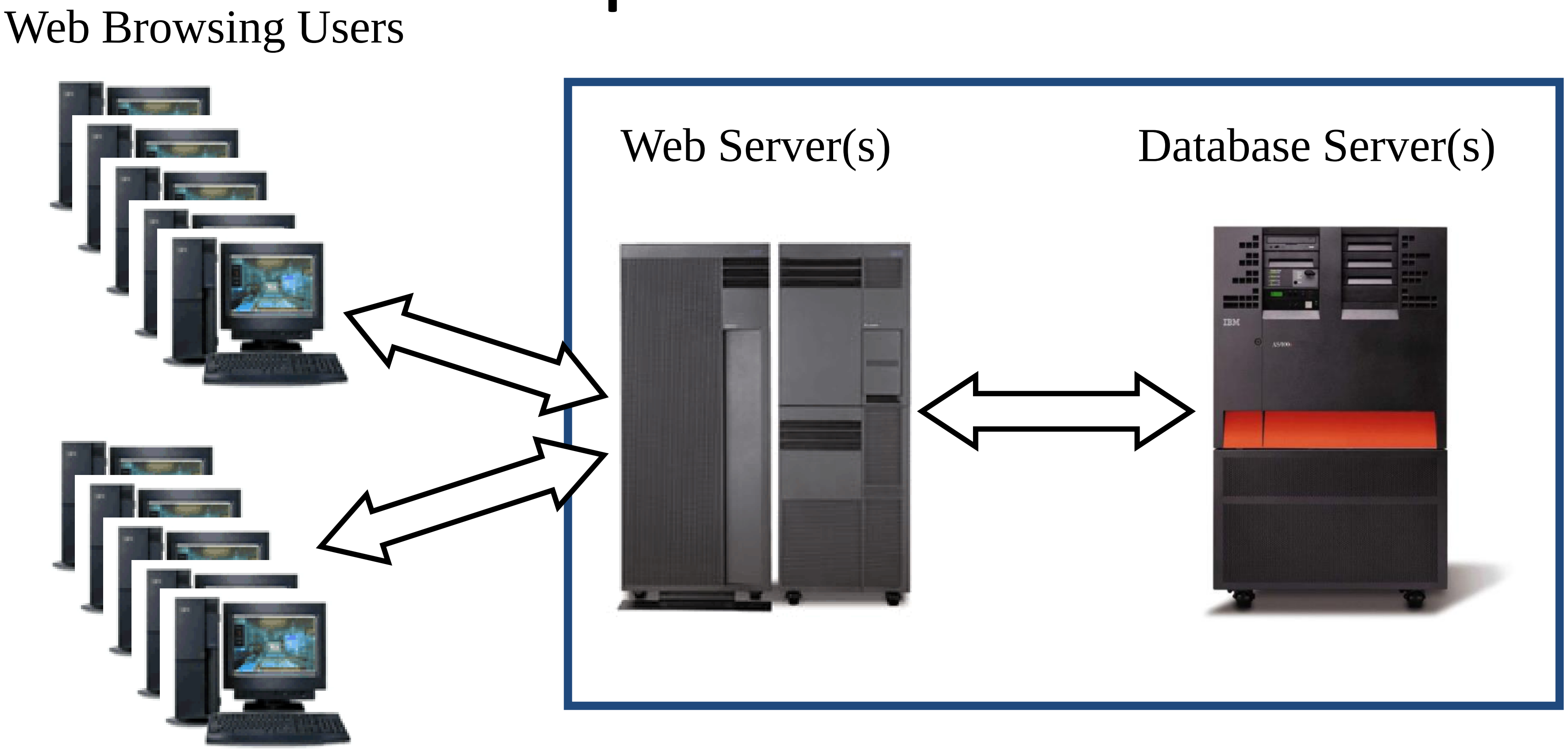
- Works well for certain application types
  - Regular control flow and memory accesses
- Difficult to guarantee correctness in all cases
  - Ambiguous memory dependences
  - Requires speculation, support for recovery
- Degree of parallelism
  - Large (1000x) for *easy* cases
  - Small (3x-10x) for *difficult* cases

3. Data Parallelism

3. Data parallelism
   - Vector, matrix, db table, pixels, web pages,…

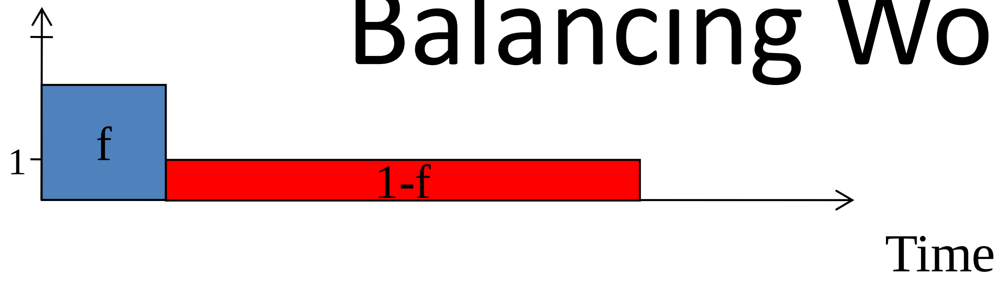
- Large data => significant parallelism
- Many ways to express parallelism
  - Vector/SIMD
  - Threads, processes, shared memory
  - Message-passing
- Challenges:
  - Balancing & coordinating work
  - Communication vs. computation at scale

4. Request Parallelism



- Multiple users => significant parallelism
- Challenges
  - Synchronization, communication, balancing work

Balancing Work



- Amdahl's parallel phase f: all processors busy
- If not perfectly balanced
  - (1-f) term grows (f not fully parallel)
  - Performance scaling suffers
- Manageable for data & request parallel apps
- Very difficult problem for other two:
  - Functional parallelism
  - Automatically extracted

Coordinating Work

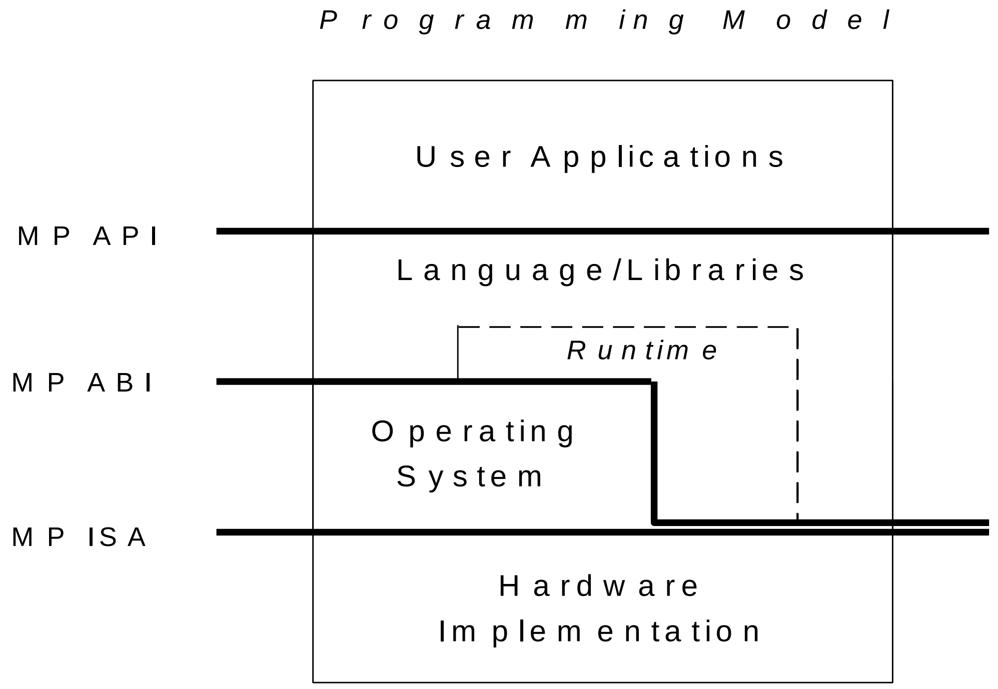
- Synchronization
  - Some data somewhere is shared
  - Coordinate/order updates and reads
  - Otherwise → chaos
- Traditionally: locks and mutual exclusion
  - Hard to get right, even harder to tune for perf.
- Research to reality: Transactional Memory
  - Programmer: Declare potential conflict
  - Hardware and/or software: speculate & check
  - Commit or roll back and retry
  - IBM, Intel, others, now support in HW

Expressing Parallelism

- SIMD – introduced by Cray-1 vector supercomputer
  - MMX, SSE/SSE2/SSE3/SSE4, AVX at small scale
- SPMD or SIMT – GPGPU model (later)
  - All processors execute same program on disjoint data
  - Loose synchronization vs. rigid lockstep of SIMD
- MIMD – most general (this lecture)
  - Each processor executes its own program
- Expressed through standard interfaces
  - API, ABI, ISA

MP Interfaces

- *Levels of abstraction* enable complex system designs (such as MP computers)
- Fairly natural extensions of uniprocessor model
  - Historical evolution



Programming Models

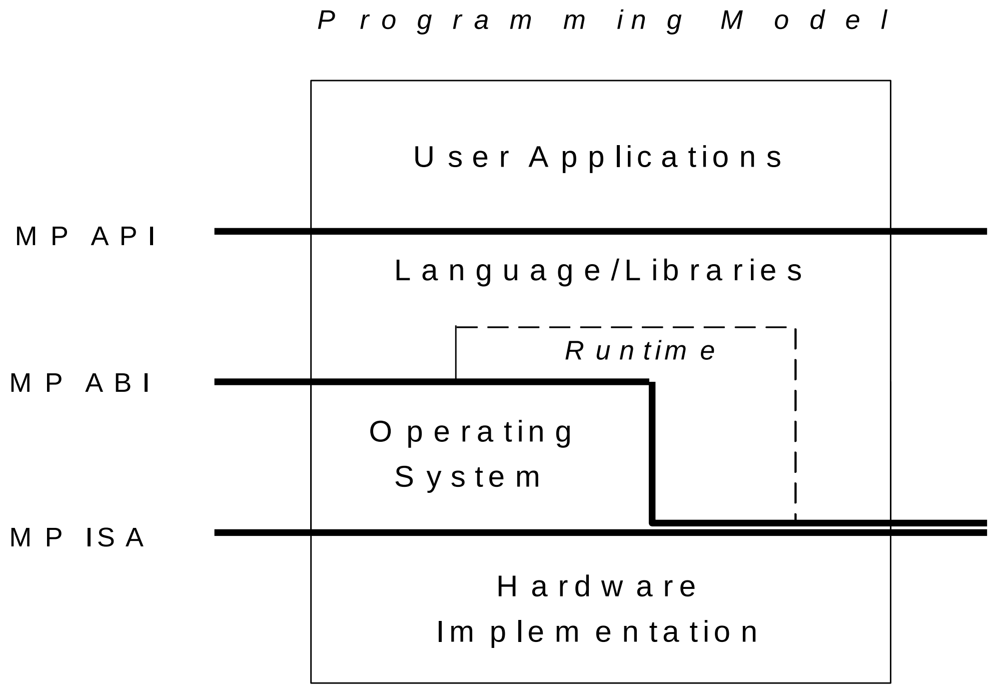
- High level paradigm for expressing an algorithm
  - Examples:
    - Functional
    - Sequential, procedural
    - Shared memory
    - Message Passing
- Embodied in high level languages that support concurrent execution
  - Incorporated into HLL constructs
  - Incorporated as libraries added to existing sequential language
- Top level features:
  - For conventional models – shared memory, message passing
  - Multiple threads are conceptually visible to programmer
  - Communication/synchronization are visible to programmer

Application Programming Interface (API)

- Interface where HLL programmer works
- High level language plus libraries
  - Individual libraries are sometimes referred to as an “API”
- User level runtime software is often part of API implementation
  - Executes procedures
  - Manages user-level state
- Examples:
  - C and pthreads
  - FORTRAN and MPI

Application Binary Interface (ABI)

- Program in API is compiled to ABI
- Consists of:
  - OS call interface
  - User level instructions (part of ISA)



Instruction Set Architecture (ISA)

- Interface between hardware and software
  - What the hardware implements
- Architected state
  - Registers
  - Memory architecture
- All instructions
  - May include parallel (SIMD) operations
  - Both non-privileged and privileged
- Exceptions (traps, interrupts)

Programming Model Elements

- For both Shared Memory and Message Passing
- Processes and threads
  - ***Process:*** A shared address space and one or more threads of control
  - ***Thread:*** A program sequencer and private address space
  - ***Task***: Less formal term – part of an overall job
  - Created, terminated, scheduled, etc.
- Communication
  - Passing of data
- Synchronization
  - Communicating control information
  - To assure reliable, deterministic communication

sub-Outline

- Shared Memory Model
  - API-level Processes, Threads
  - API-level Communication
  - API-level Synchronization
- Shared Memory Implementation
  - Implementing Processes, Threads at ABI/ISA levels
  - Implementing Communication at ABI/ISA levels
  - Implementing Synchronization at ABI/ISA levels

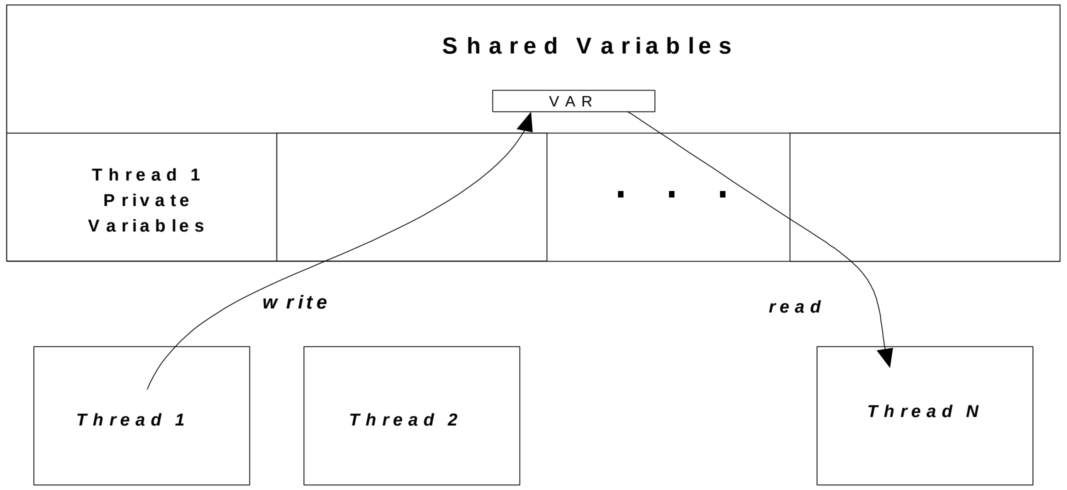
In order of decreasing complexity:

synchronization, processes&threads, communication

- Repeat the above for Message Passing

Shared Memory

- Flat shared memory or object heap
  - Synchronization via memory variables enables reliable sharing
- Single process
- Multiple threads per process
  - Private memory per thread
- Typically built on shared memory hardware system



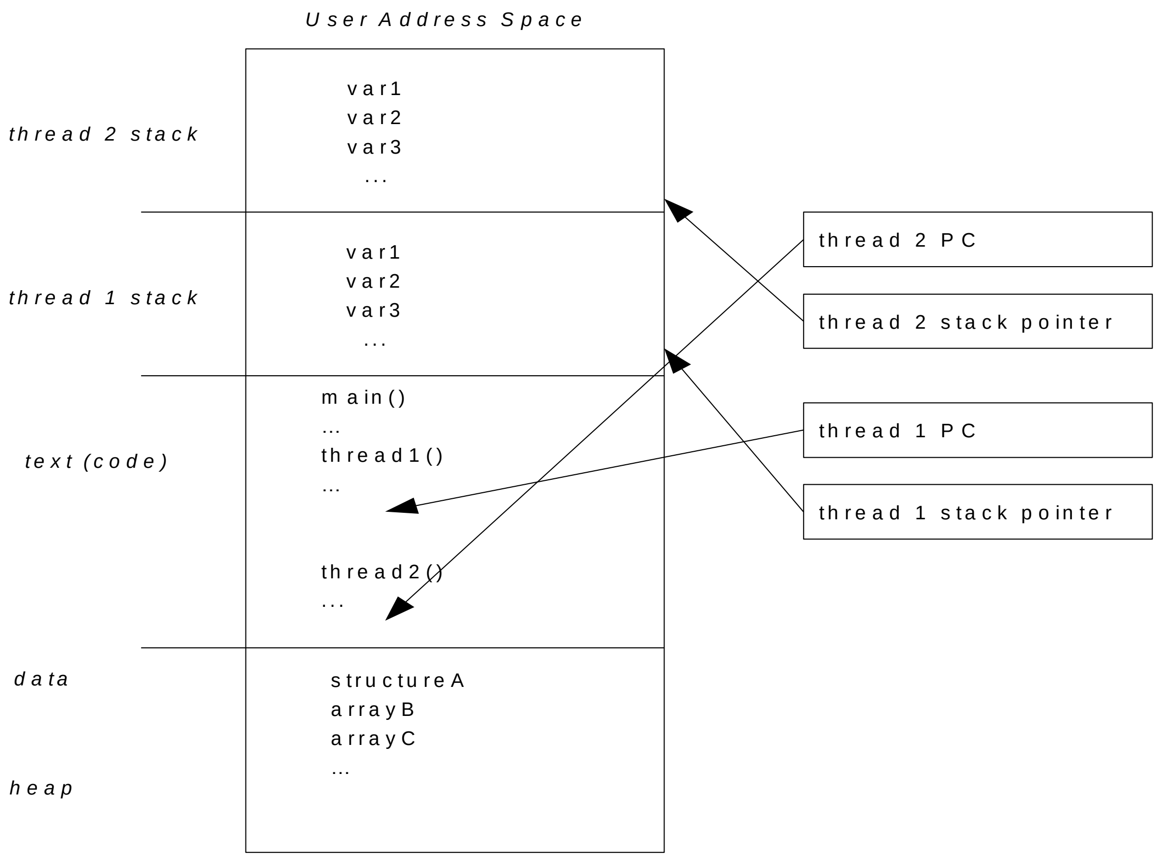
Threads and Processes

- Creation
  - generic -- Fork
    - (Unix forks a process, not a thread)
  - pthread_create(….*thread_function….)
    - creates new thread in current address space
- Termination
  - pthread_exit
    - or terminates when thread_function terminates
  - pthread_kill
    - one thread can kill another

Example

- Unix process with two threads

  (PC and stack pointer actually part of ABI/ISA implementation)



Shared Memory Communication

Thread 0	Thread 1
	load r1, A
	addi r1, r1, 3
load r1, A	
addi r1, r1, 1	
store r1, A	
	store r1, A

(a)

Thread 0	Thread 1
load r1, A	
addi r1, r1, 1	
store r1, A	
	load r1, A
	addi r1, rl, 3
	store r1, A

(b)

Thread 0	Thread 1
	load r1, A
	addi r1, r1, 3
	store r1, A
load r1, A	
addi r1, r1, 1	
store r1, A	

(c)

Thread 0	Thread 1
load r1, A	
addi r1, r1, 1	
	load r1, A
	addi r1, rl, 3
	store r1, A
store r1, A	

(d)

- Reads and writes to shared variables via normal language (assignment) statements (e.g. assembly load/store)

Shared Memory Synchronization

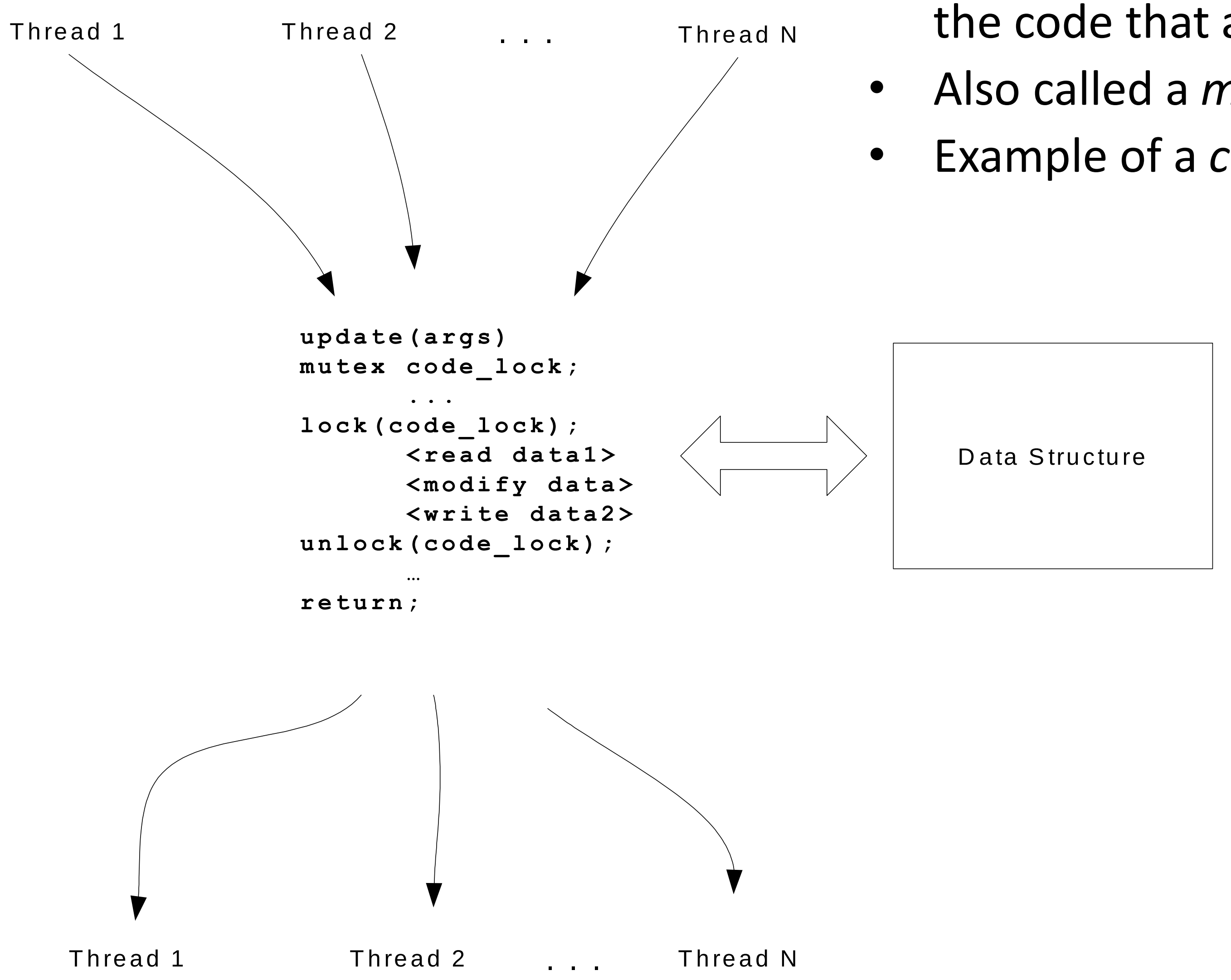
- What really gives shared memory programming its structure
- Usually explicit in shared memory model
  - Through language constructs or API
- Three major classes of synchronization
  - Mutual exclusion (mutex)
  - Point-to-point synchronization
  - Rendezvous
- Employed by *application design patterns*
  - *A general description or template for the solution to a commonly recurring software design problem.*

Mutual Exclusion (mutex)

- Assures that only one thread at a time can access a code or data region
- Usually done via *locks*
  - One thread acquires the lock
  - All other threads excluded until lock is released
- Examples
  - pthread_mutex_lock
  - pthread_mutex_unlock
- Two main application programming patterns
  - Code locking
  - Data locking

Code Locking

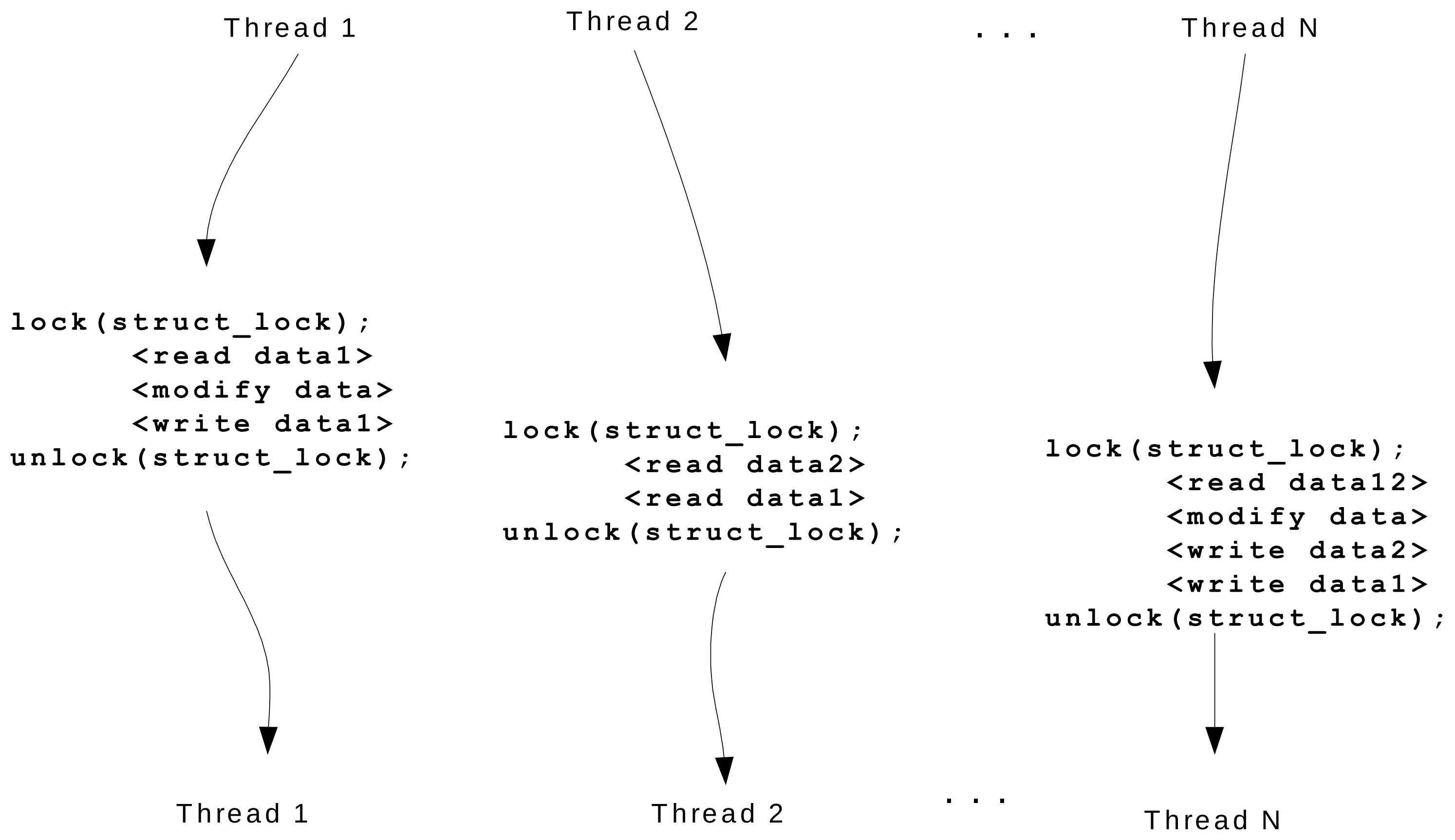
- Protect shared data by locking the code that accesses it
- Also called a *monitor* pattern
- Example of a *critical section*



```
update(args)
mutex code_lock;
      . . .
lock(code_lock);
      <read data1>
      <modify data>
      <write data2>
unlock(code_lock);
      …
return;
```

Data Locking

- Protect shared data by locking data structure



```
lock(struct_lock);
      <read data1>
      <modify data>
      <write data1>
unlock(struct_lock);
```

```
lock(struct_lock);
      <read data2>
      <read data1>
unlock(struct_lock);
```

```
lock(struct_lock);
      <read data12>
      <modify data>
      <write data2>
      <write data1>
unlock(struct_lock);
```

Data Locking

- Preferred when data structures are read/written in combinations
- Example:

```
<thread 0>
Lock(mutex_struct1)
Lock(mutex_struct2)
    <access struct1>
    <access struct2>
Unlock(mutex_data1)
Unlock(mutex_data2)
```

```
<thread 1>
Lock(mutex_struct1)
Lock(mutex_struct3)
    <access struct1>
    <access struct3>
Unlock(mutex_data1)
Unlock(mutex_data3)
```

```
<thread 2>
Lock(mutex_struct2)
Lock(mutex_struct3)
    <access struct2>
    <access struct3>
Unlock(mutex_data2)
Unlock(mutex_data3)
```

Deadlock

- Data locking is prone to deadlock
  - If locks are acquired in an unsafe order
- Example:

<thread 0>

```
Lock(mutex_data1)
Lock(mutex_data2)
   <access data1>
   <access data2>
Unlock(mutex_data1)
Unlock(mutex_data2)
```

<thread 1>

```
Lock(mutex_data2)
Lock(mutex_data1)
   <access data1>
   <access data2
Unlock(mutex_data1)
Unlock (mutex_data2)
```

- Complexity
  - Disciplined locking order must be maintained, else deadlock
  - Also, composability problems
    - Locking structures in a nest of called procedures

Efficiency

- Lock Contention
  – Causes threads to wait
- Function of lock *granularity*
  – Size of data structure or code that is being locked
- Extreme Case:
  – "One big lock" model for multithreaded OSes
  – Easy to implement, but very inefficient
- Finer granularity
  + Less contention
  - More locks, more locking code
  - perhaps more deadlock opportunities
- Coarser granularity
  – opposite +/- of above

Point-to-Point Synchronization

- One thread signals another that a condition holds
  - Can be done via API routines
  - Can be done via normal load/stores
- Examples
  - pthread_cond_signal
  - pthread_cond_wait
    - suspends thread if condition not true
- Application program pattern
  - Producer/Consumer

<Producer>

```
while (full ==1){}; wait
buffer = value;
full = 1;
```

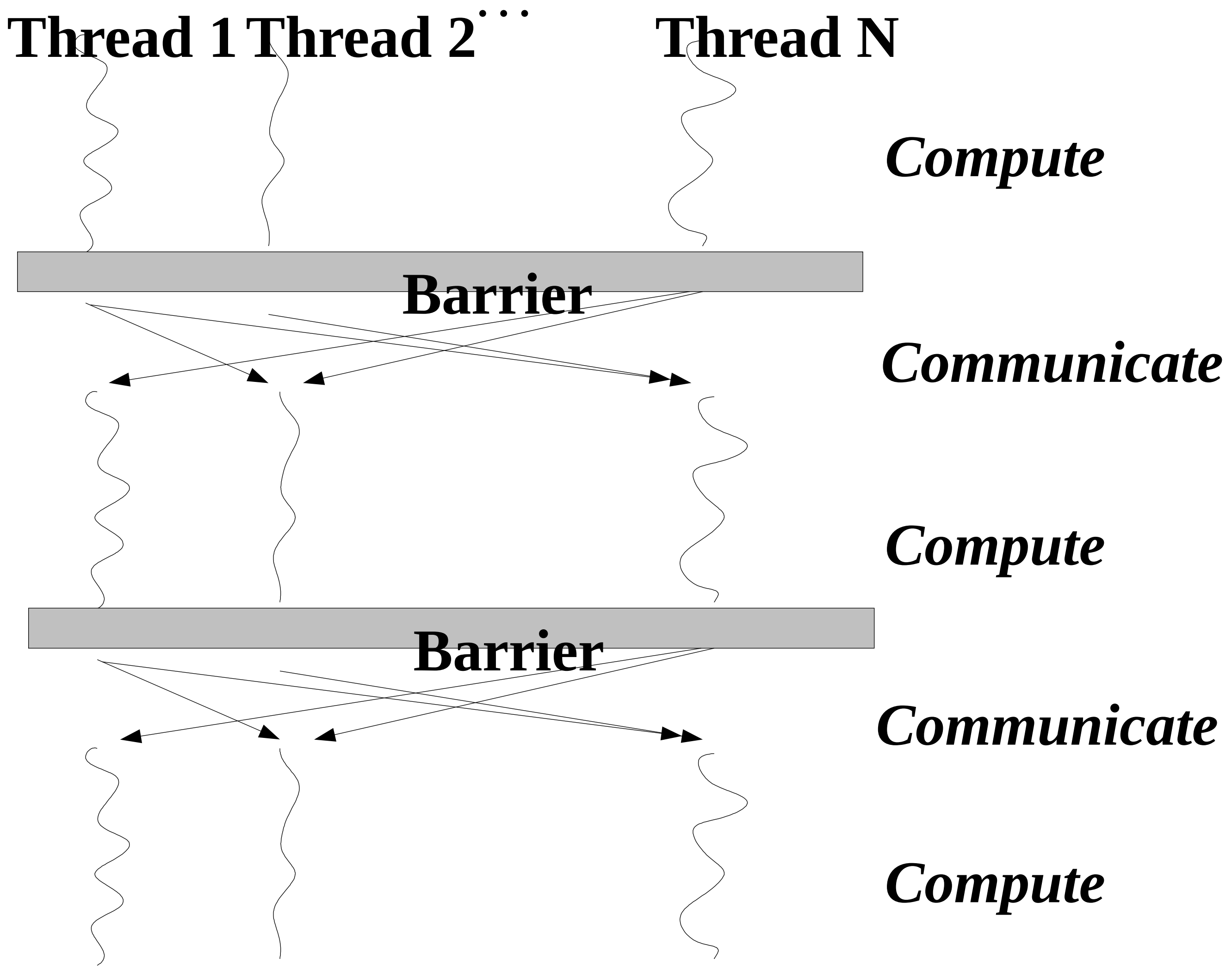
<Consumer>

```
while (full == 0){}; wait
b = buffer;
full = 0;
```

Rendezvous

- Two or more cooperating threads must reach a program point before proceeding
- Examples
  - wait for another thread at a join point before proceeding
    - example: pthread_join
  - barrier synchronization
    - many (or all) threads wait at a given point
- Application program pattern
  - Bulk synchronous programming pattern

Bulk Synchronous Program Pattern



Summary: Synchronization and Patterns

- mutex (mutual exclusion)
  - code locking (monitors)
  - data locking
- point to point
  - producer/consumer
- rendezvous
  - bulk synchronous

sub-Outline

- Shared Memory Model
  - API-level Processes, Threads
  - API-level Communication
  - API-level Synchronization
- Shared Memory Implementation
  - Implementing Processes, Threads at ABI/ISA levels
  - Implementing Communication at ABI/ISA levels
  - Implementing Synchronization at ABI/ISA levels

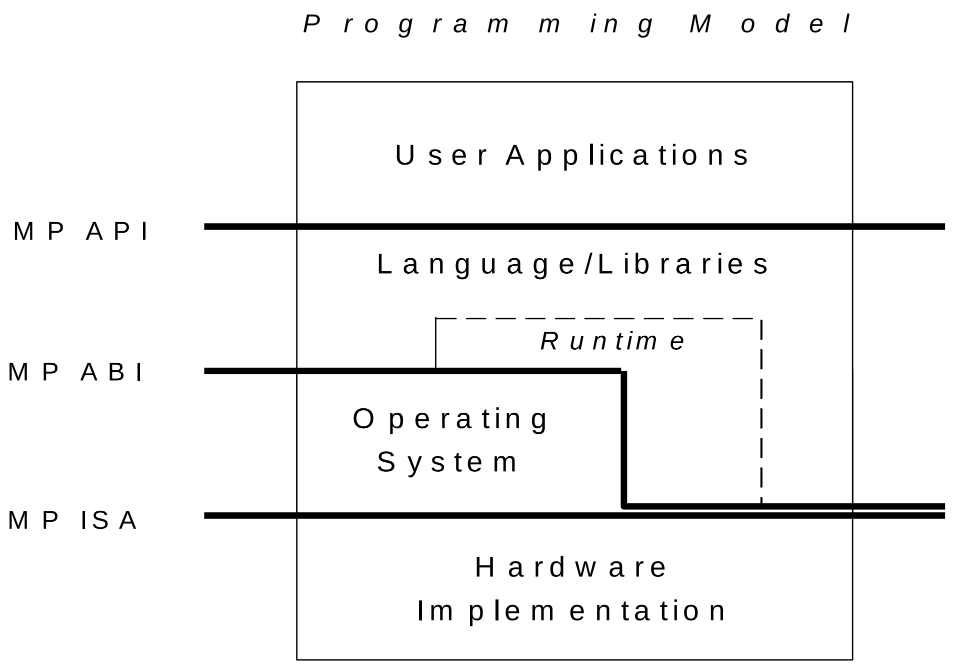
In order of decreasing complexity:

synchronization, processes&threads, communication

- Repeat the above for Message Passing

API Implementation

- Implemented at ABI and ISA level
  - OS calls
  - Runtime software
  - Special instructions



Processes and Threads

- Three models
  - OS processes
  - OS threads
  - User threads

OS Processes

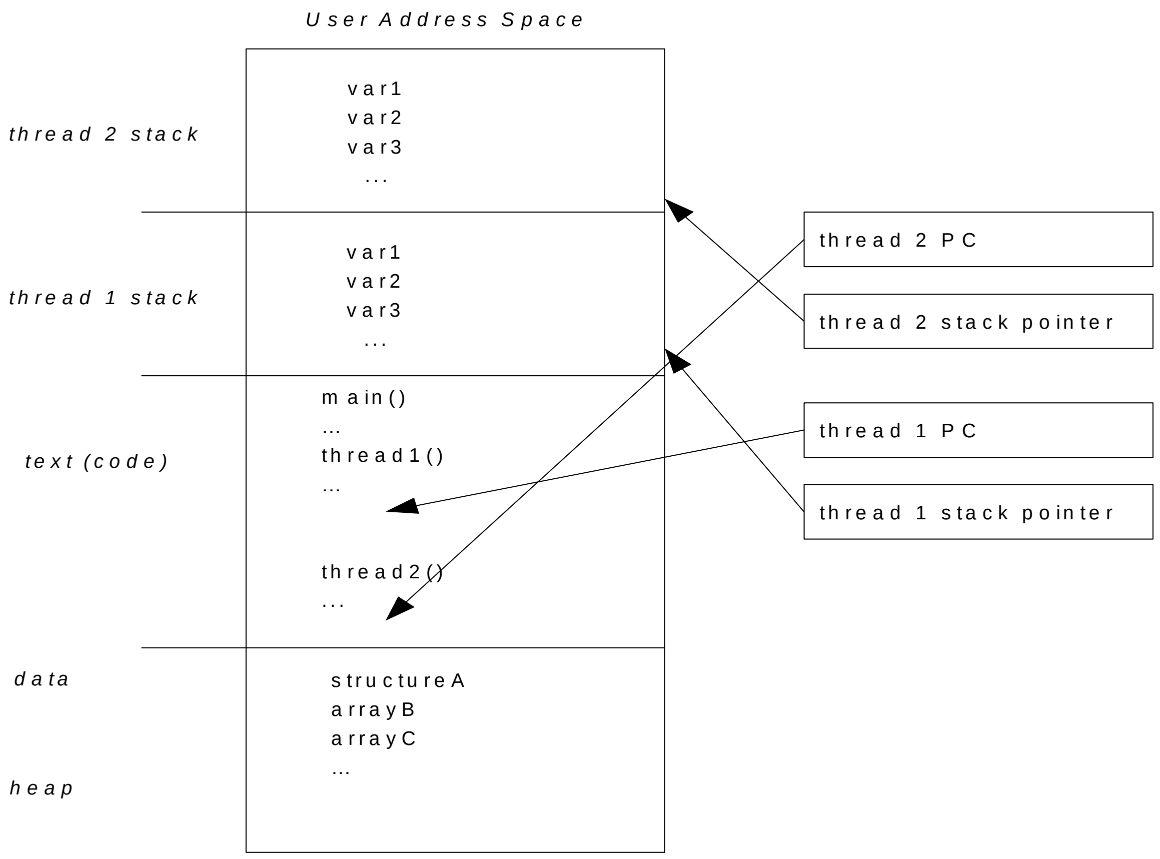
- Thread == Process
- Use OS fork to create processes
- Use OS calls to set up shared address space (e.g. shmget)
- OS manages processes (and threads) via run queue
- Heavyweight thread switches
  - OS call followed by:
  - Switch address mappings
  - Switch process-related tables
  - Full register switch
- Advantage
  - Threads have protected private memory

OS (Kernel) Threads

- API pthread_create() maps to Linux clone()
  - Allows multiple threads sharing memory address space
- OS manages threads via run queue
- Lighter weight thread switch
  - Still requires OS call
  - No need to switch address mappings
  - OS switches architected register state and stack pointer

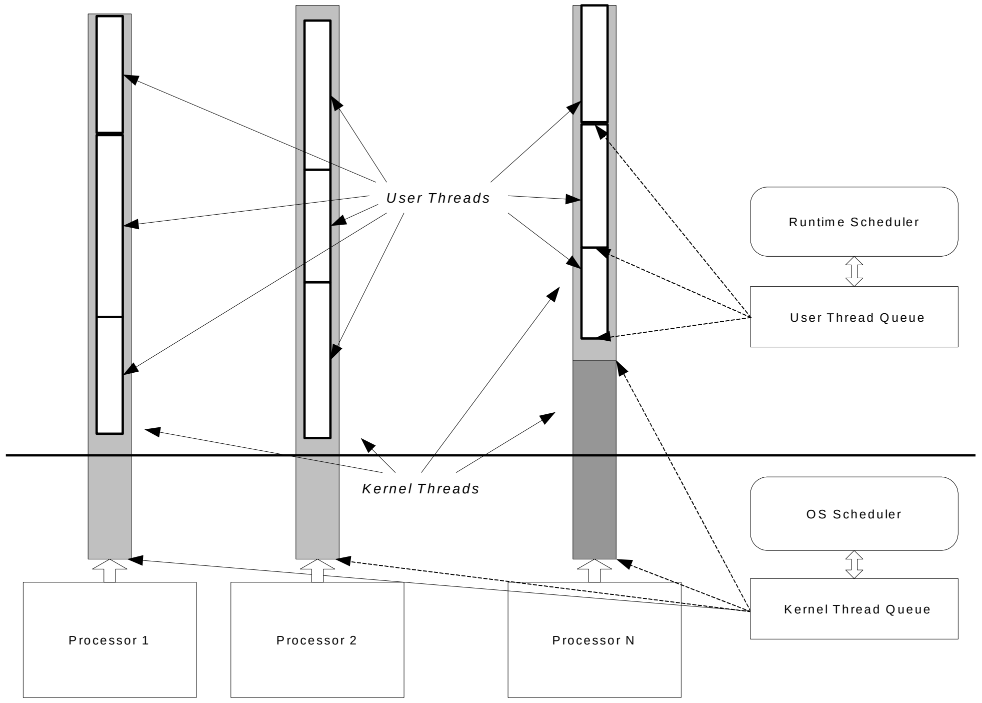
User Threads

- If memory mapping doesn't change, why involve OS at all?
- Runtime creates threads simply by allocating stack space
- Runtime switches threads via user level instructions
  - thread switch via jumps

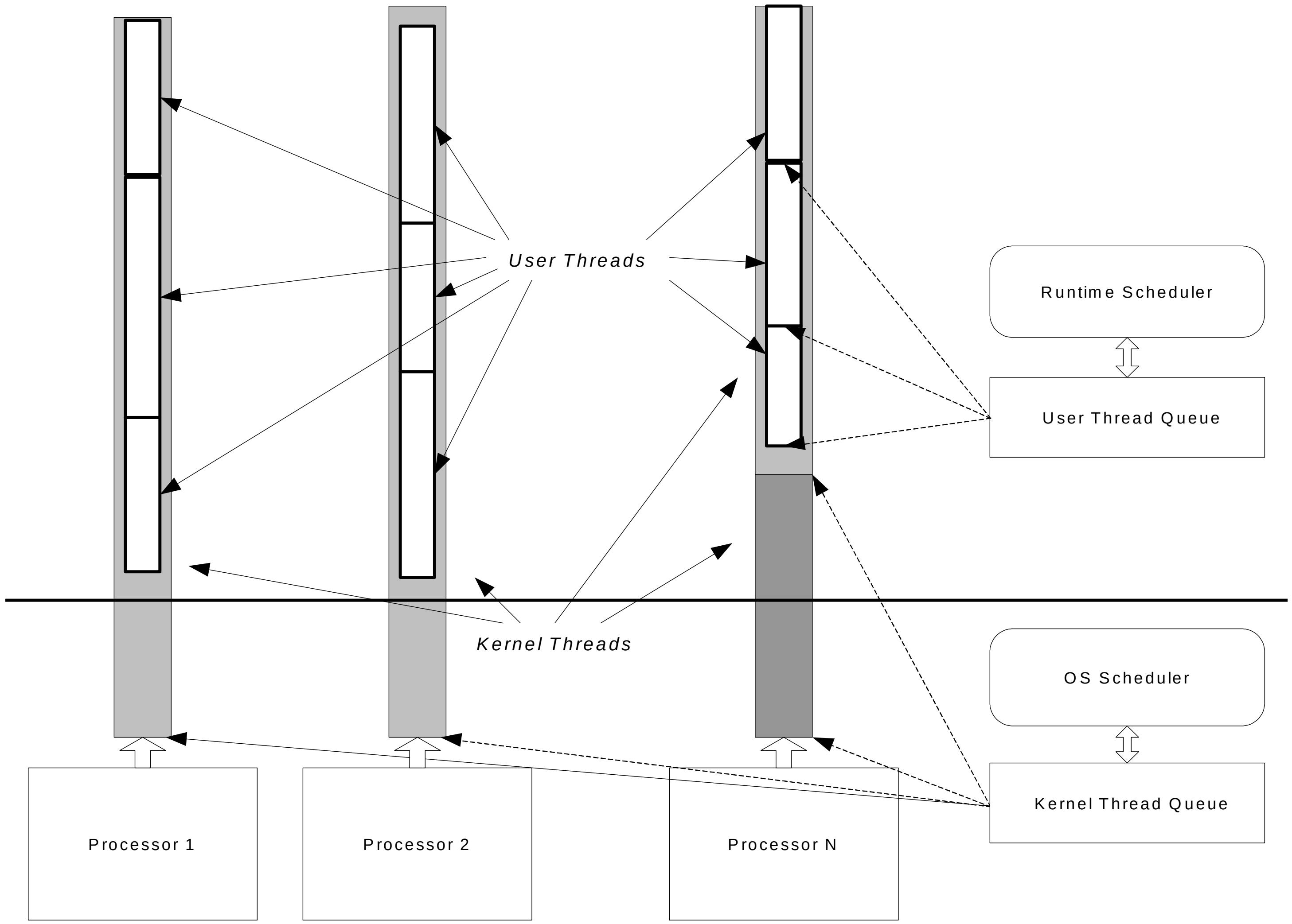


Implementing User Threads

- Multiple kernel threads needed to get control of multiple hardware processors
- Create kernel threads (OS schedules)
- Create user threads that runtime schedules onto kernel threads



Implementing User Threads



Communication

- *Easy*

  Just map high level access to variables to ISA level loads and stores
- *Except for*

  Ordering of memory accesses -- later

Synchronization

- Implement locks and rendezvous (barriers)
- Use loads and stores to implement lock:

```
        <thread 0>                          <thread 1>
                  .                                    .
                  .                                    .
LAB1:             Load R1, Lock             LAB2:      Load R1, Lock
                  Branch LAB1 if R1==1                 Branch LAB2 if R1==1
                  Ldi R1, 1                            Ldi R1,1
                  Store Lock, R1                       Store Lock, R1
                  .                                    .
                  <critical section>                   <critical section>
                  .                                    .
                  Ldi R1, 0                            Ldi R1, 0
                  Store  Lock, R1                      Store Lock, R1
```

Lock Implementation

- *Does not work*
- Violates mutual exclusion if both threads attempt to lock at the same time
  - In practice, may work *most* of the time…
  - Leading to an unexplainable system hang every few days

<thread 0>

```
        .
        .
LAB1:   Load R1, Lock
        Branch LAB1 if R1==1
        Ldi R1, 1
        Store Lock, R1
```

<thread 1>

```
        .
        .
LAB2:   Load R1, Lock
        Branch LAB2 if R1==1
        Ldi R1,1
        Store Lock, R1
```

Lock Implementation

- Reliable locking can be done with *atomic* read-modify-write instruction
- Example: test&set
  - read lock and write a one
  - some ISAs also set CCs (test)

<thread 1>

```
                .
LAB1:           Test&Set R1, Lock
                Branch LAB1 if R1==1
                .
                <critical section>
                .
                Reset Lock
```

<thread 2>

```
                .
LAB2:   Test&Set R1, Lock
        Branch LAB2 if R1==1
        .
        <critical section>
        .
        Reset Lock
```

Atomic Read-Modify-Write

- Many such instructions have been used in ISAs

```
Test&Set(reg,lock)
reg ←mem(lock);
mem(lock) ← 1;
```

```
Fetch&Add(reg,value,sum)
reg ← mem(sum);
mem(sum)←mem(sum)+value;
```

```
Swap(reg,opnd)
temp←mem(opnd);
mem(opnd)← reg;
reg ← temp
```

- More-or-less equivalent
  - One can be used to implement the others
  - Implement Fetch&Add with Test&Set:

```
try:   Test&Set(lock);
       if lock == 1 go to try;
       reg ←mem(sum);
       mem(sum) ← reg+value;
       reset (lock);
```

Sub-Atomic Locks

- Use two instructions:
  Load linked + Store conditional
  - Load linked
    - reads memory value
    - sets special flag
    - writes address to special global address register
  - Flag cleared on
    - operations that may violate atomicity
      - (implementation-dependent)
      - e.g., write to address by another processor
      - can use cache coherence mechanisms (later)
    - context switch
  - Store conditional
    - writes value if flag is set
    - no-op if flag is clear
    - sets CC indicating or failure

Load-Linked Store-Conditional

- Example: atomic swap (r4,mem(r1))

```
try: mov    r3,r4        ;move exchange value
     ll     r2,0(r1)     ;load locked
     sc     r3,0(r1)     ;store conditional
     beqz   r3,try       ;if store fails
     mov    r4,r2        ;load value to r4
```

- RISC- style implementation
  - Like many early RISC ideas, it seemed like a good idea at the time…

    register windows, delayed branches, special divide regs, etc.

Lock Efficiency

- Spin Locks
  - tight loop until lock is acquired

```
LAB1:     Test&Set R1, Lock
          Branch LAB1 if R1==1
```

- Inefficiencies:
  - Memory/Interconnect resources, spinning on read/writes
  - With a cache-based systems,

    writes ⇒ lots of coherence traffic
  - Processor resource
    - not executing useful instructions

Efficient Lock Implementations

- Test&Test&Set
  - spin on check for unlock only, then try to lock
  - with cache systems, all reads can be local
    - no bus or external memory resources used

```
test_it:  load          reg, mem(lock)
          branch        test_it if reg==1
lock_it:  test&set      reg, mem(lock)
          branch        test_it if reg==1
```

- Test&Set with Backoff
  - Insert delay between test&set operations (not too long)
  - Each failed attempt ⇒ longer delay
    (Like ethernet collision avoidance)

Efficient Lock Implementations

- Solutions just given save memory/interconnect resource
  - Still waste processor resource
- Use runtime to suspend waiting process
  - Detect lock
  - Place on wait queue
  - Schedule another thread from run queue
  - When lock is released move from wait queue to run queue

Point-to-Point Synchronization

- *Can* use normal variables as flags

```
while (full ==1){}  ;spin      while (full == 0){} ;spin
a = value;                     b = value;
full = 1;                      full = 0;
```

- Assumes sequential consistency (later)
  - Using normal variables may cause problems with relaxed consistency models
- May be better to use special opcodes for flag set/clear

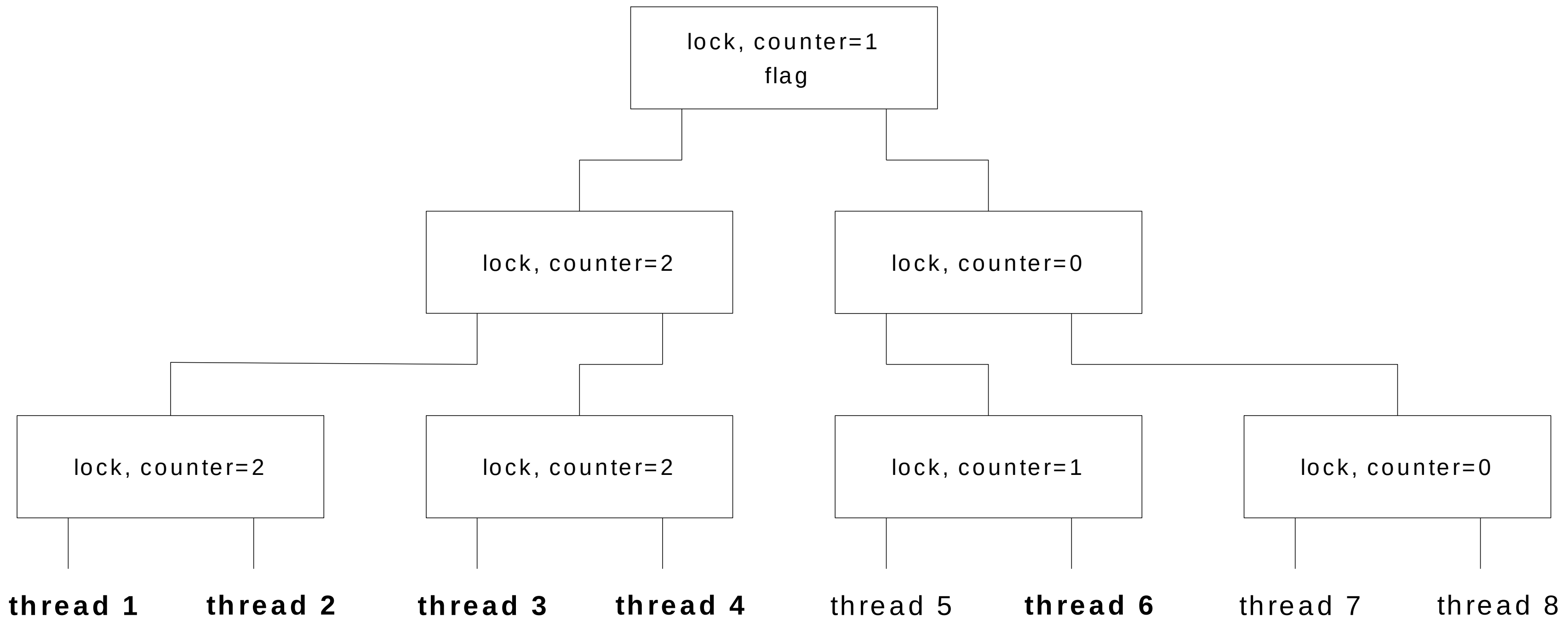
Barrier Synchronization

- Uses a lock, a counter, and a flag
  - lock for updating counter
  - flag indicates all threads have incremented counter

```
Barrier (bar_name, n) {
    Lock (bar_name.lock);
    if (bar_name.counter = 0)  bar_name.flag = 0;
    mycount = bar_name.counter++;
    Unlock (bar_name.lock);
    if (mycount == n) {
            bar_name.counter = 0;
            bar_name.flag = 1;
    }
    else  while(bar_name.flag = 0) {};        /* busy wait */
}
```

Scalable Barrier Synchronization

- Single counter can be point of contention
- Solution: use tree of locks
- Example:
  - threads 1,2,3,4,6 have completed



Memory Ordering

- Program Order
  - Processor executes instructions in architected (PC) sequence

    or at least appears to
- Loads and stores from a single processor execute in *program order*
  - Program order *must* be satisfied
  - It is part of the ISA
- What about ordering of loads and stores from *different* processors

Memory Ordering

- Producer/Consumer example:

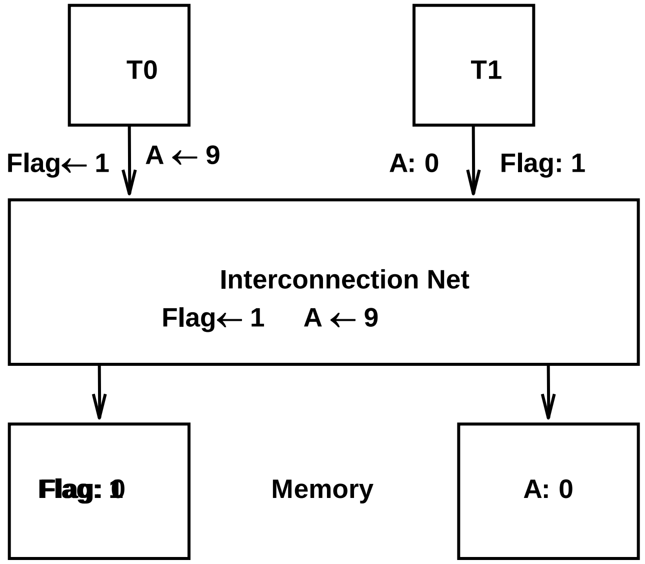
```
T0:       A=0;                   T1:
          Flag = 0;
          ....
          A=9;                             While (Flag==0){};
          Flag = 1;              L2:       if (A==0)...
```

- Intuitively it is *impossible* for A to be 0 at L2
  - *But* it can happen if the updates to memory are reordered by the memory system
- In an MP system, memory ordering rules must be carefully defined and maintained

Practical Implementation

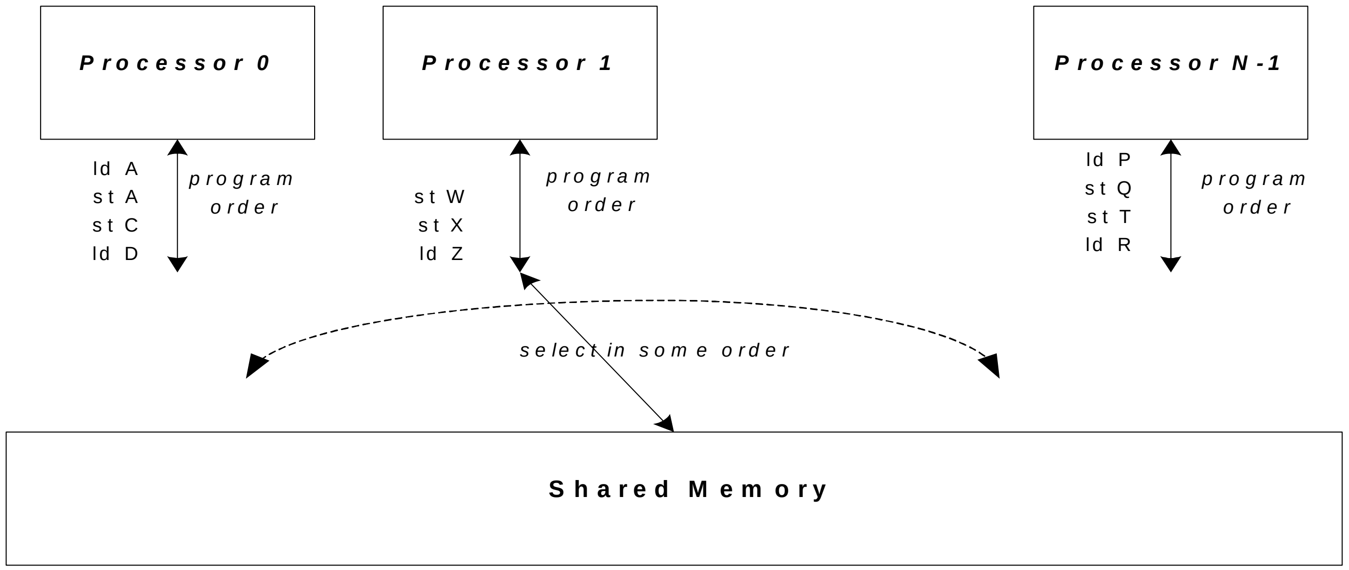
- Interconnection network with contention and buffering

```
T0:    A=0;                 T1:
       Flag = 0;
       ....
       A=9;                      While (Flag==0){};
       Flag = 1;                 if (A==0)...
```



Sequential Consistency

"A system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order and the operations of each individual processor appears in this sequence in the order specified by its program“ -- Leslie Lamport



Memory Coherence

- WRT individual memory locations, consistency is always maintained
  - In producer/consumer examples, coherence is always maintained
- Practically, memory coherence often reduces to cache coherence
  - Cache coherence implementations to be discussed later
- Summary
  - *Coherence* is for a single memory location
  - *Consistency* applies to apparent ordering of all memory locations
  - Memory coherence and consistency are ISA concepts

```
Thread0:             Thread1:
Store A←0
Store A←9
                     Load A=9
```

(a)

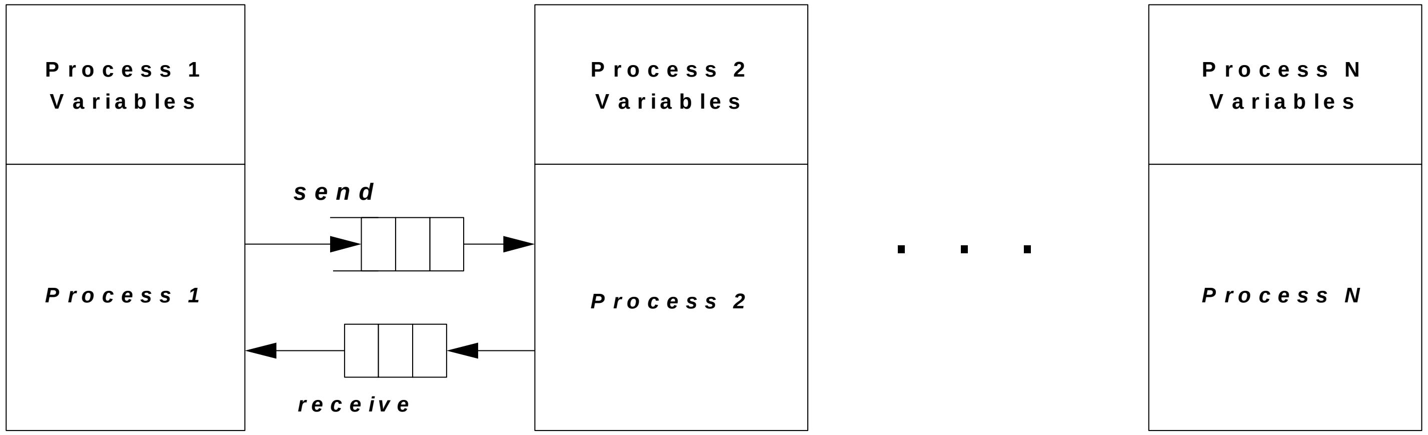
```
Thread0:             Thread1:
Store Flag←0
                     Load Flag=0
                     Load Flag=0
Store Flag←1
                     Load Flag=1
```

(b)

sub-Outline

- Message Passing Model
  - API-level Processes, Threads
  - API-level Communication
  - API-level Synchronization
- Message Passing Implementation
  - Implementing Processes, Threads at ABI/ISA levels
  - Implementing Communication at ABI/ISA levels
  - Implementing Synchronization at ABI/ISA levels

Message Passing



- Multiple processes (or threads)
- Logical data partitioning
  - No shared variables
- Message Passing
  - Threads of control communicate by sending and receiving messages
  - May be implicit in language constructs
  - More commonly explicit via API

MPI –Message Passing Interface API

- A widely used standard
  - For a variety of distributed memory systems
    - SMP Clusters, workstation clusters, MPPs, heterogeneous systems
- Also works on Shared Memory MPs
  - Easy to emulate distributed memory on shared memory HW
- Can be used with a number of high level languages

Processes and Threads

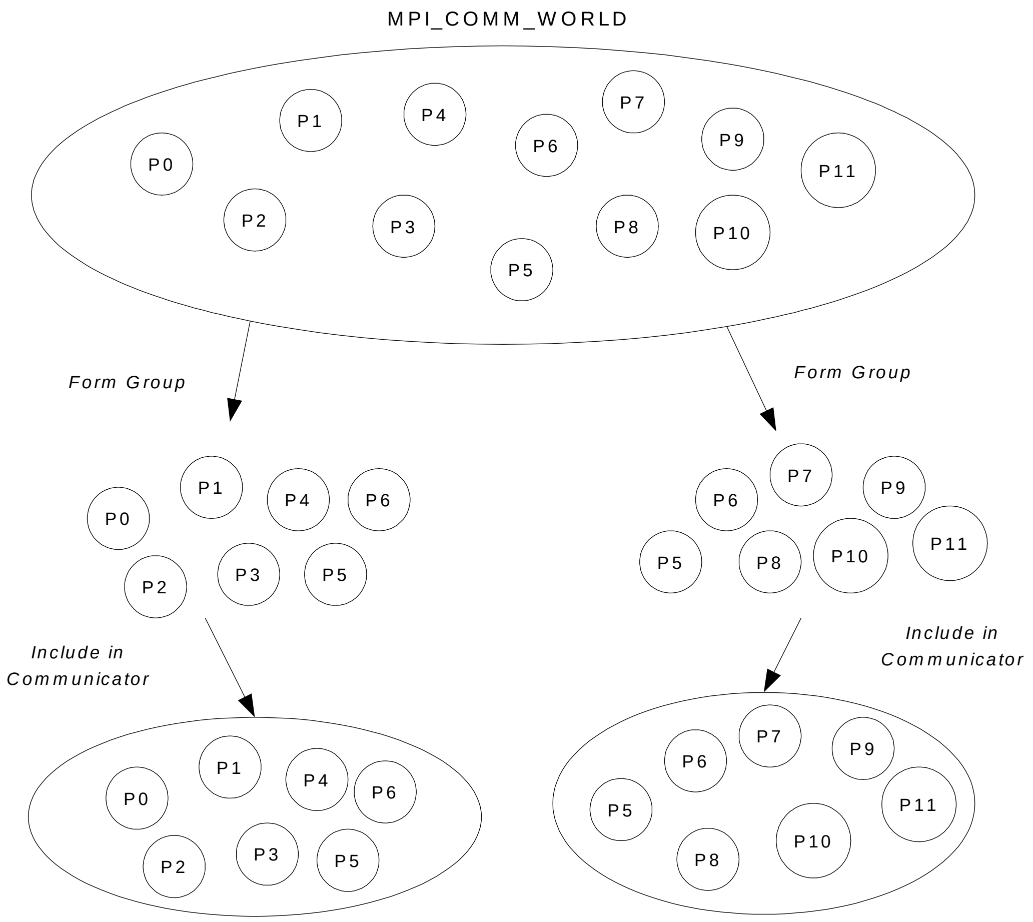
- Lots of flexibility (advantage of message passing)
  - 1) Multiple threads sharing an address space
  - 2) Multiple processes sharing an address space
  - 3) Multiple processes with different address spaces
    - and different OSes
- 1 and 2 are easily implemented on shared memory hardware (with single OS)
  - Process and thread creation/management similar to shared memory
- 3 probably more common in practice
  - Process creation often external to execution environment; e.g. shell script
  - Hard for user process on one system to create process on another OS

Process Management

- Processes are given identifiers (PIds)
  - "rank" in MPI
- Process can acquire own PId
- Operations can be conditional on PId
- Message can be sent/received via PIds

Process Management

- Organize into groups
  - For collective management and communication



Communication and Synchronization

- Combined in the message passing paradigm
  - Synchronization of messages part of communication semantics
- Point-to-point communication
  - From one process to another
- Collective communication
  - Involves groups of processes
  - e.g., broadcast

Point to Point Communication

- Use sends/receives
- send(RecProc, SendBuf,…)
  - RecProc is destination (wildcards may be used)
  - SendBuf names buffer holding message to be sent
- receive (SendProc, RecBuf,…)
  - SendProc names sending process (wildcards may be used)
  - RecBuf names buffer where message should be placed

MPI Examples

- MPI_Send(buffer,count,type,dest,tag,comm)
  - buffer – address of data to be sent
  - count – number of data items
  - type – type of data items
  - dest – rank of the receiving process
  - tag – arbitrary programmer-defined identifier
    - tag of send and receive must match
  - comm – communicator number
- MPI_Recv(buffer,count,type,source,tag,comm,status)
  - buffer – address of data to be sent
  - count – number of data items
  - type – type of data items
  - source – rank of the sending process; may be a wildcard
  - tag – arbitrary programmer-defined identifier; may be a wildcard
    - tag of send and receive must match
  - comm – communicator number
  - status – indicates source, tag, and number of bytes transferred

Message Synchronization

- After a send or receive is executed…
  - *Has message actually been sent? or received?*
- Asynchronous versus Synchronous
  - Higher level concept
- Blocking versus non-Blocking
  - Lower level – depends on buffer implementation
    - *but is reflected up into the API*

Synchronous vs Asynchronous

- Synchronous Send
  - Stall until message has actually been received
  - Implies a message acknowledgement from receiver to sender
- Synchronous Receive
  - Stall until message has actually been received
- Asynchronous Send and Receive
  - Sender and receiver can proceed regardless
  - Returns *request handle* that can be tested for message receipt
  - Request handle can be tested to see if message has been sent/received

Asynchronous Send

- MPI_Isend(buffer,count,type,dest,tag,comm,request)
  - buffer – address of data to be sent
  - count – number of data items
  - type – type of data items
  - dest – rank of the receiving process
  - tag – arbitrary programmer-defined identifier
    - tag of send and receive must match
  - comm – communicator number
  - request – a unique number that can be used later to test for completion (via Test or Wait)
- Sending process is immediately free to do other work
- Must test *request* handle before another message can be safely sent
  - **MPI_test** – tests request handle # and returns status
  - **MPI_wait** – blocks until request handle# is “done”

Asynchronous Receive

- MPI_Irecv(buffer,count,type,source,tag,comm,request)

  buffer – address of data to be sent

  count – number of data items

  type – type of data items

  source – rank of the sending process; may be a wildcard

  tag – arbitrary programmer-defined identifier; may be a wildcard

  tag of send and receive must match

  comm – communicator number

  request – a unique number that can be used later to test for completion
- Receiving process does not wait for message
- Must test *request* handle before message is known to be in buffer
  - **MPI_test** – tests request handle # and returns status
  - **MPI_wait** – blocks until request handle# is "done"

MPI Example: Comm. Around a Ring

```
int main(argc,argv)
int argc;
char *argv[];
{
int numprocs, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
prev = rank-1;
next = rank+1;
if (rank == 0) prev = numprocs - 1;
if (rank == (numprocs - 1)) next = 0;
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);
MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);
MPI_Waitall(4, reqs, stats);
MPI_Finalize();
 }
```

Deadlock

- Blocking communications may deadlock

```
<Process 0>                       <Process 1>
Send(Process1, Message);          Send(Process0, Message);
Receive(Process1, Message);       Receive(Process0, Message);
```

- Requires careful (safe) ordering of sends/receives

```
<Process 0>                       <Process 1>
Send(Process1, Message);          Receive (Process0, Message);
Receive(Process1, Message);       Send (Process0, Message);
```

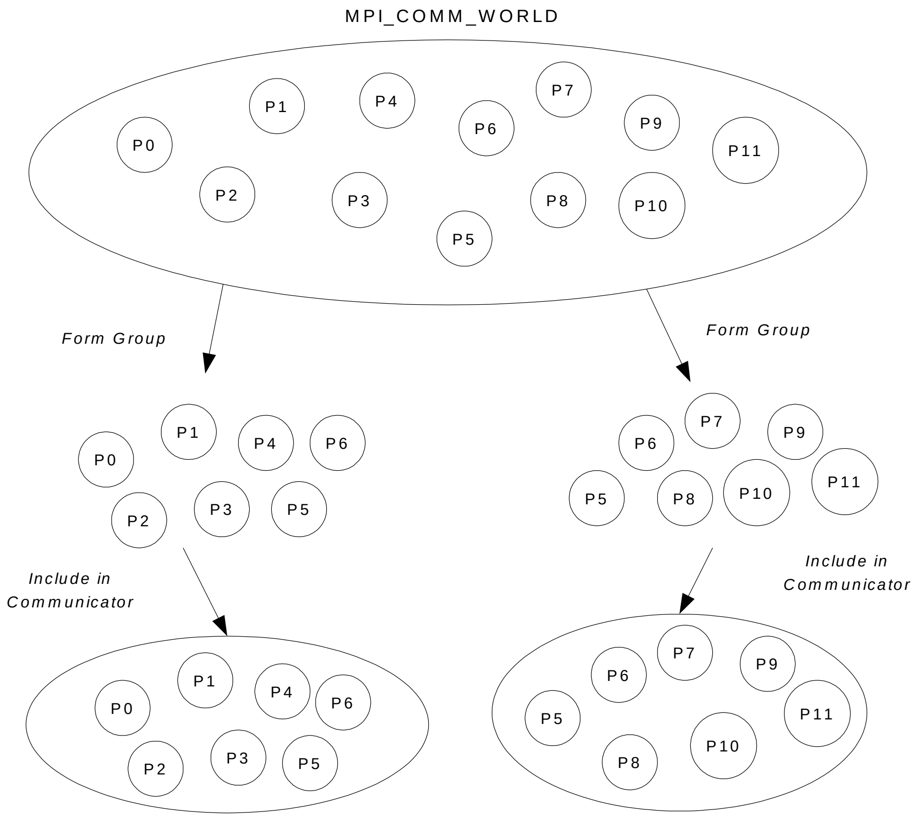
- Also depends on buffering
  - System buffering may not eliminate deadlock, just postpone it

Collective Communications

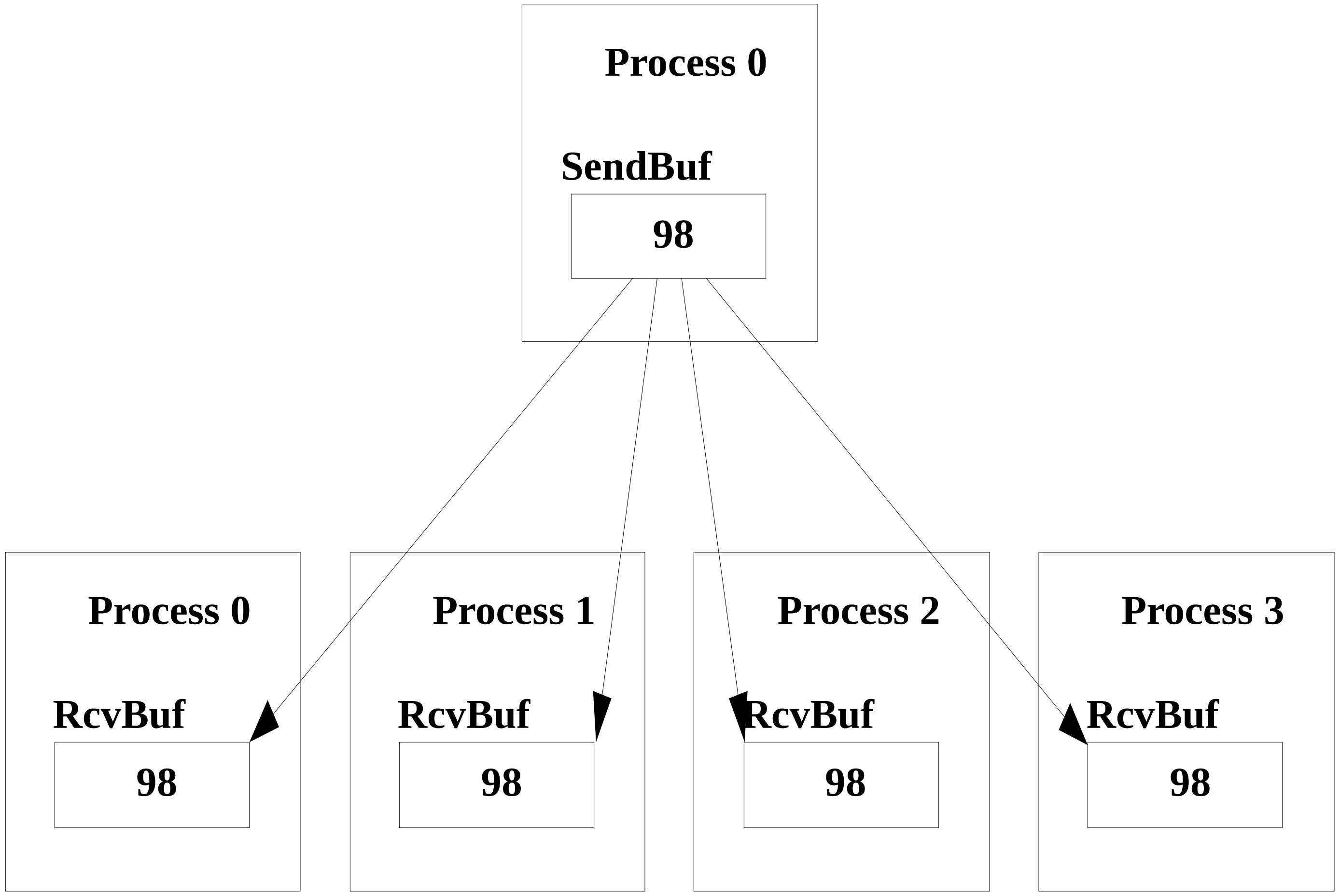
- Involve all processes within a communicator
- Blocking
- MPI_Barrier (comm)
  - Barrier synchronization
- MPI_Bcast (*buffer,count,datatype,root,comm)
  - Broadcasts from process of rank “root” to all other processes
- MPI_Scatter (*sendbuf,sendcnt,sendtype,*recvbuf,
  ...... recvcnt,recvtype,root,comm)
  - Sends different messages to each process in a group
- MPI_Gather (*sendbuf,sendcnt,sendtype,*recvbuf,
  ...... recvcount,recvtype,root,comm)
  - Gathers different messages from each process in a group
- Also reductions

Communicators and Groups

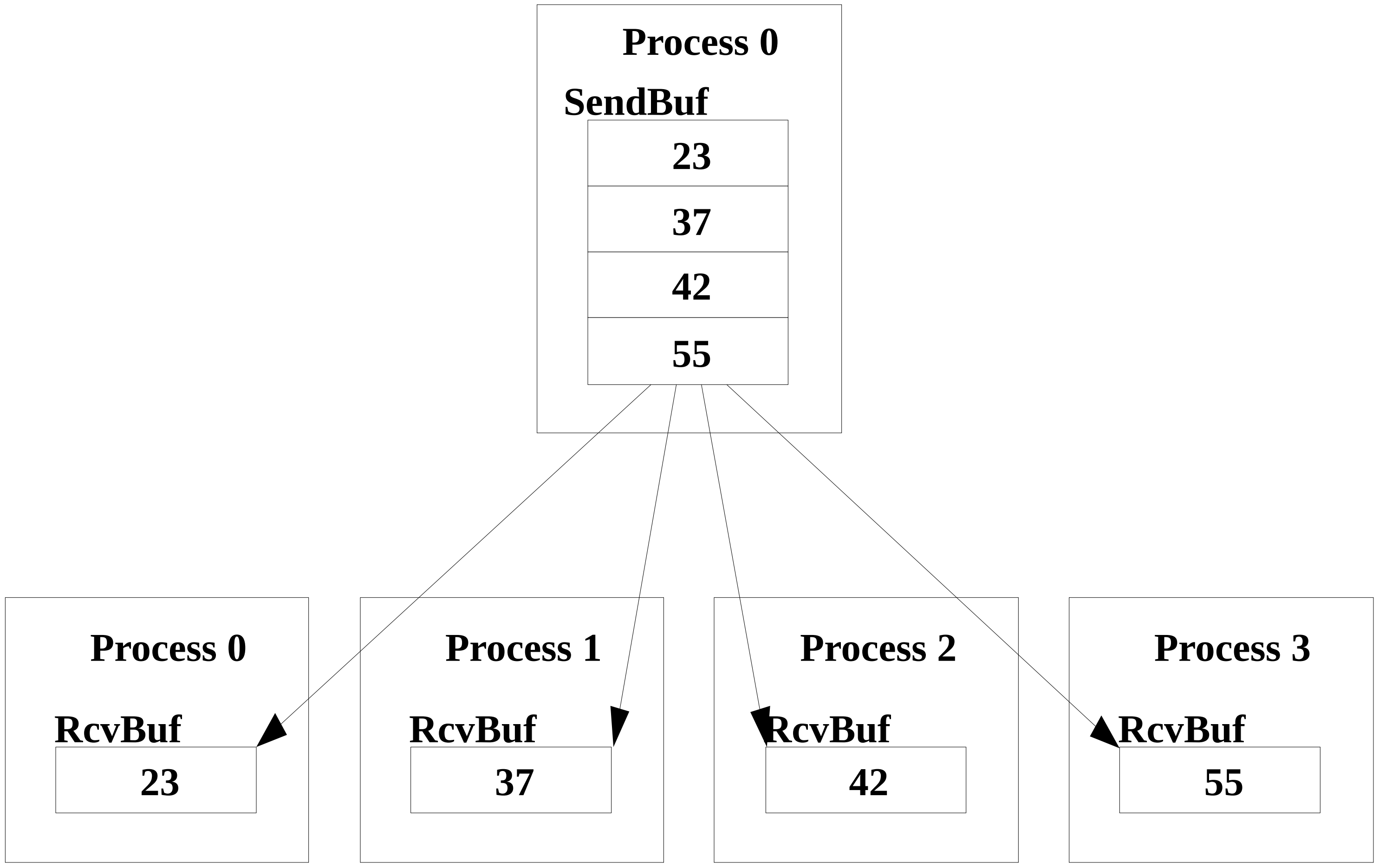
- Define collections of processes that may communicate
  - Often specified in message argument
  - MPI_COMM_WORLD – predefined communicator that contains all processes



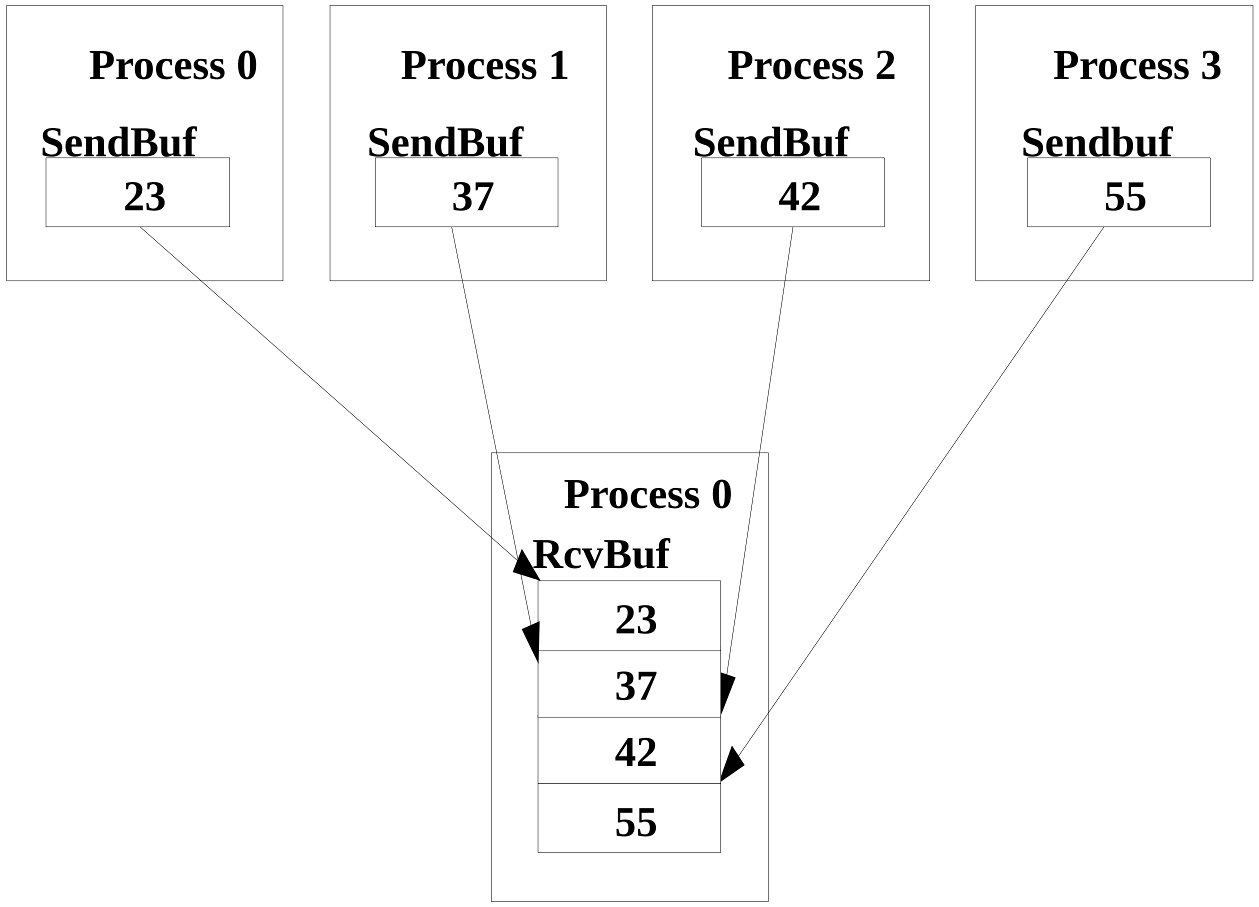
Broadcast Example



Scatter Example



Gather Example



Message Passing Implementation

- At the ABI and ISA level
  - No special support (beyond that needed for shared memory)
  - Most of the implementation is in the runtime
    - user-level libraries
  - Makes message passing relatively portable
- Three implementation models (given earlier)

  1) Multiple threads sharing an address space

  2) Multiple processes sharing an address space

  3) Multiple processes with non-shared address space
     and different OSes

Multiple Threads Sharing Address Space

- Runtime manages buffering and tracks communication
  - Communication via normal loads and stores using shared memory
- Example: Send/Receive
  - Send calls runtime, runtime posts availability of message in runtime-managed table
  - Receive calls runtime, runtime checks table, finds message
  - Runtime copies data from send buffer to store buffer via load/stores
- Fast/Efficient Implementation
  - May even be advantageous over shared memory paradigm
    - considering portability, software engineering aspects
  - Can use runtime thread scheduling
  - Problem with protecting private memories and runtime data area

Multiple Processes Sharing Address Space

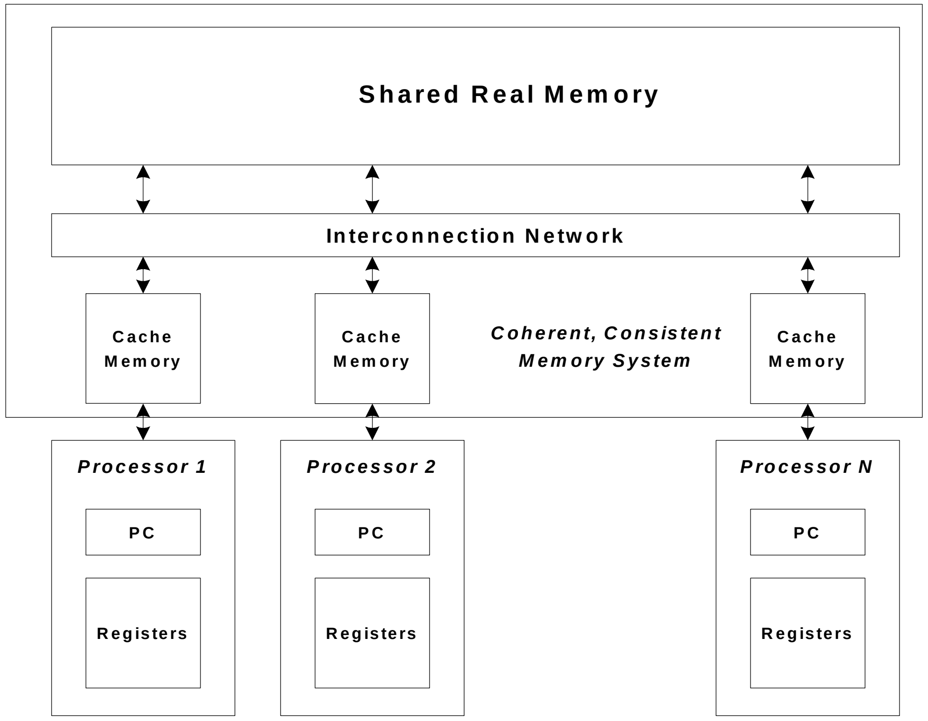
- Similar to multiple threads sharing address space
- Would rely on kernel scheduling
- May offer more memory protection
  - With intermediate runtime buffering
  - User processes can not access others' private memory

Multiple Processes with Non-Shared Address Space

- Most common implementation
- Communicate via networking hardware
- Send/receive to runtime
  - Runtime converts to OS (network) calls
- Relatively high overhead
  - Most HPC systems use special low-latency, high-bandwidth networks
  - Buffering in receiver's runtime space may save some overhead for receive (doesn't require OS call)

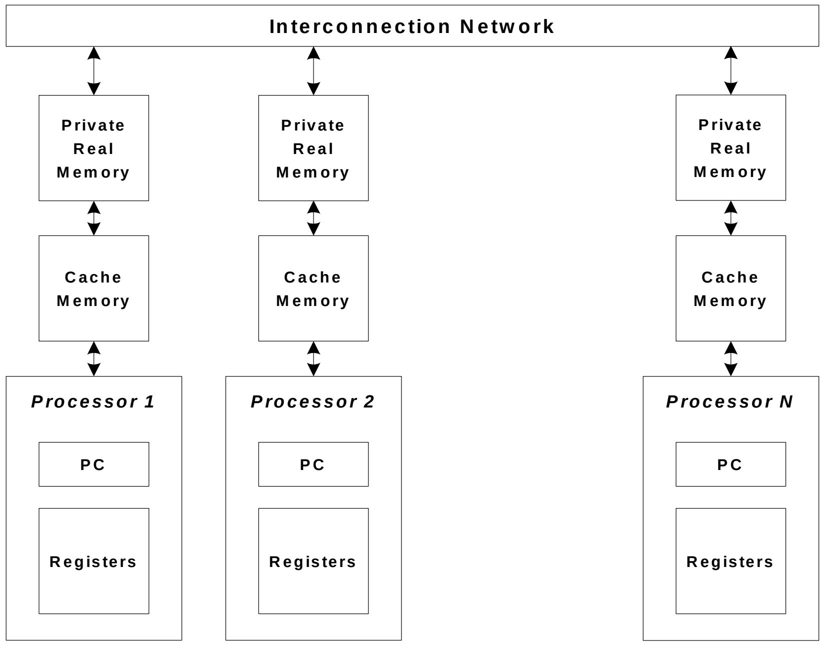
At the ISA Level: Shared Memory

- Multiple processors
- Architected shared virtual memory
- Architected Synchronization instructions
- Architected Cache Coherence
- Architected Memory Consistency



At the ISA Level: Message Passing

- Multiple processors
- Shared or non-shared real memory (multi-computers)
- Limited ISA support   (if any)
  - An advantage of distributed memory systems --Just connect a bunch of small computers
  - Some implementations may use shared memory managed by runtime



Lecture Summary

- Introduction to Parallel Software
- Programming Models
- Major Abstractions
  - Processes & threads
  - Communication
  - Synchronization
- Shared Memory
  - API description
  - Implementation at ABI, ISA levels
  - ISA support
- Message Passing
  - API description
  - Implementation at ABI, ISA levels
  - ISA support
- Not covered: openmp, intel tbb, CUDA (later), etc.