Unit 2

Synchronization and Data Sharing

What Is Parallel Processing?

- Parallel processing
  - Also called Multiprocessing
  - Two or more CPUs execute instructions simultaneously
  - Processor Manager
    - Coordinates activity of each CPU
    - Synchronizes interaction among CPUs

What Is Parallel Processing? (continued)

- Parallel processing development
  - Enhances throughput
  - Increases computing power
- Benefits
  - Increased reliability
    - More than one CPU
    - If one CPU fails, others take over
    - Not simple to implement
  - Faster processing
    - Instructions processed in parallel two or more at a time

What Is Parallel Processing? (continued)

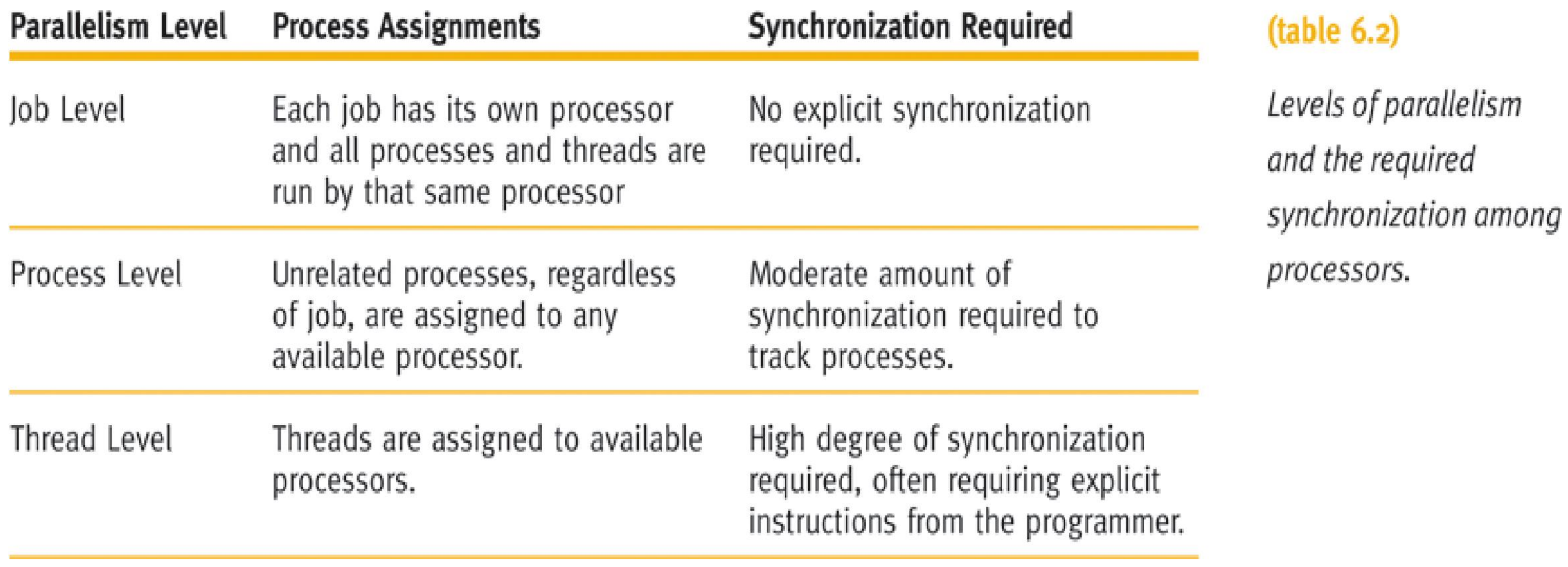
- Faster instruction processing methods
  - CPU allocated to each program or job
  - CPU allocated to each working set or parts of it
  - Individual instructions subdivided
    - Each subdivision processed simultaneously
    - This is also called **Concurrent programming**
- Two major challenges
  - Connecting processors into configurations
  - Orchestrating processor interaction

Synchronization is key to the system's success in a parallel processing environment.

Evolution of Multiprocessors

- Developed for high-end midrange and mainframe computers
  - Each additional CPU treated as additional resource
- Today hardware costs reduced
  - Multiprocessor systems available on all systems
- Multiprocessing occurs at three levels
  - Job level
  - Process level
  - Thread level
    - Each requires different synchronization frequency

Evolution of Multiprocessors (continued)



Parallelism Level	Process Assignments	Synchronization Required
Job Level	Each job has its own processor and all processes and threads are run by that same processor	No explicit synchronization required.
Process Level	Unrelated processes, regardless of job, are assigned to any available processor.	Moderate amount of synchronization required to track processes.
Thread Level	Threads are assigned to available processors.	High degree of synchronization required, often requiring explicit instructions from the programmer.

(table 6.2)

Levels of parallelism and the required synchronization among processors.

Introduction to Multi-Core Processors

- Multi-core processing
  - Several processors placed on single chip
- Problems
  - Heat and current leakage (tunneling)
- Solution
  - Single chip with two processor cores in same space
    - Allows two sets of simultaneous calculations
    - 80 or more cores on single chip
  - Two cores each run more slowly than single core chip

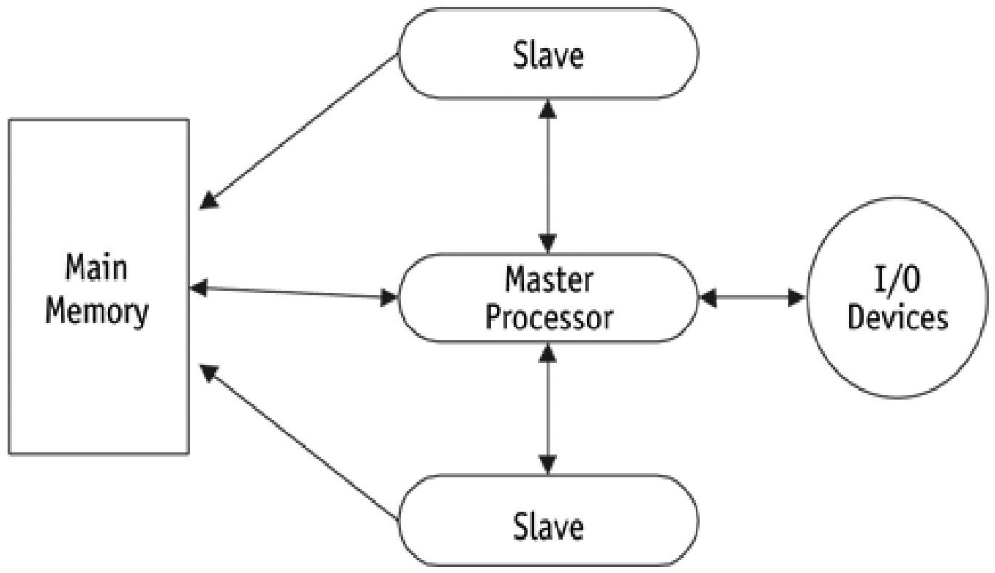
Typical Multiprocessing Configurations

- Multiple processor configuration impacts systems
- Three types
  - Master/slave
  - Loosely coupled
  - Symmetric

Master/Slave Configuration

- Asymmetric multiprocessing system
- Single-processor system with additional slave processors
    - Each slave processor managed by the primary master processor
- Master processor responsibilities
  - Manages entire system: files, I/O devices, memory, and CPUs
  - Maintains status of all processes in the system
  - Performs storage management activities
  - Schedules work for other processors
  - Executes all control programs

Master/Slave Configuration (continued)



(figure 6.1)

In a master/slave multiprocessing configuration, slave processors send all I/O requests through the master processor.

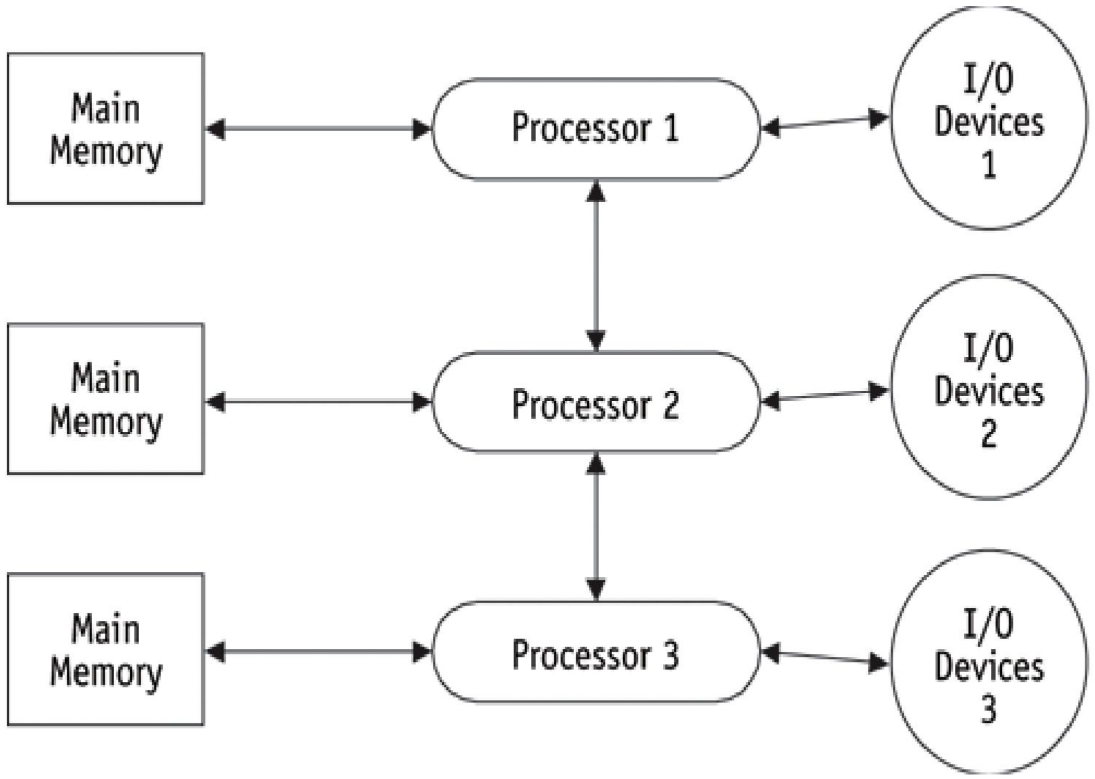
Master/Slave Configuration (continued)

- Advantages
  - Simplicity
- Disadvantages
  - Reliability
    - No higher than single processor system
  - Potentially poor resources usage
  - Increases number of interrupts

Loosely Coupled Configuration

- Several complete computer systems
  - Each with its own memory, I/O devices, CPU, and OS
    - Maintains its own commands and I/O management tables
- Difference between loosely coupled system and a collection of independent single-processing systems is that:
  - Each processor
    - Communicates and cooperates with others
    - Has **global tables** which indicate to which CPU each job has been allocated
- **Job scheduling** based on policies such as new jobs assigned to CPU with lightest load or with the best combination of I/O devices available
- Even is a single processor failure occurs:
  - Others continue work independently

Loosely Coupled Configuration (continued)



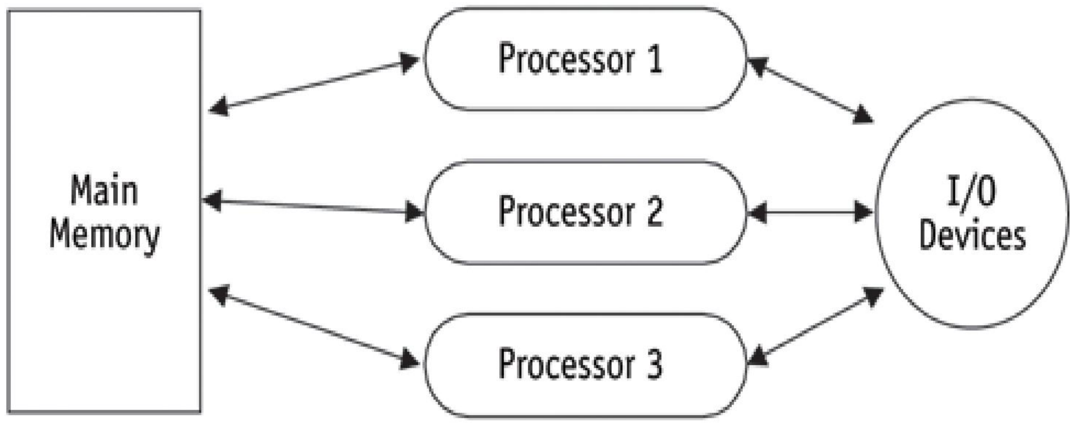
(figure 6.2)

In a loosely coupled multiprocessing configuration, each processor has its own dedicated resources.

Symmetric or Tightly Coupled Configuration

- Decentralized process scheduling
  - Single operating system copy
  - Global table listing
- Interrupt processing
  - Update corresponding process list
  - Run another process
- More conflicts
  - Several processors access same resource at same time
- **Process synchronization**
  - Algorithms resolving conflicts between processors

Symmetric or Tightly Coupled Configuration



(figure 6.3)

A symmetric multiprocessing configuration with homogeneous processors. Processes must be carefully synchronized to avoid deadlocks and starvation.

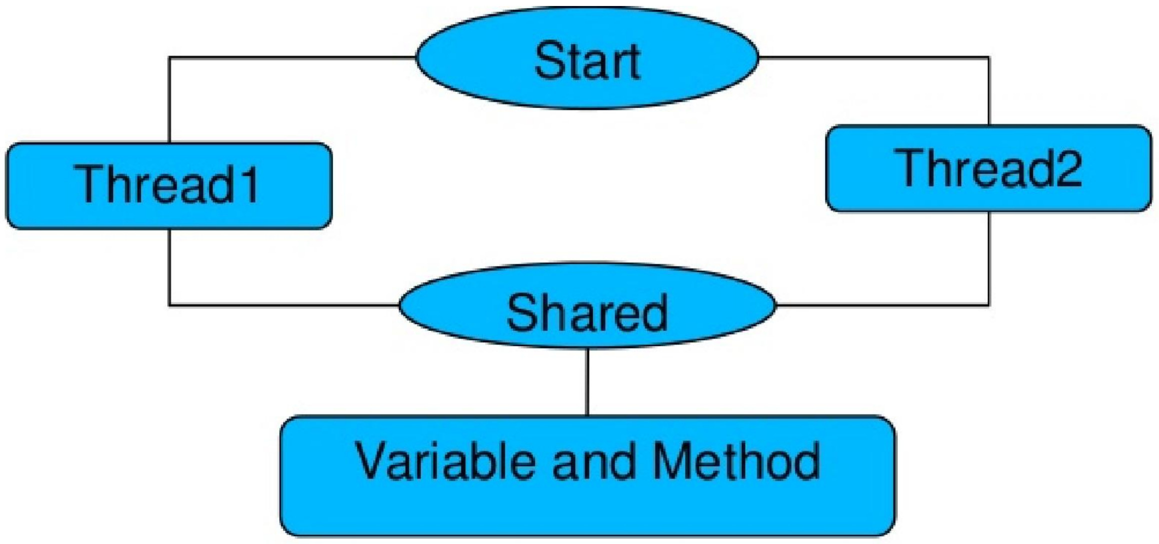
Symmetric or Tightly Coupled Configuration

- Advantages (over loosely coupled configuration)
  - Uses resources effectively
  - Can balance loads well
  - Can degrade gracefully in failure situation
- Most difficult to implement
  - Requires well synchronized processes
    - Avoids races and deadlocks

Thread

- The threads that are executed independently to each other are called as asynchronous threads.
- Problems:
  - 2 or more threads share the same resource while only one of them can access the resource at one time.
  - If the producer and consumer are sharing the same kind of data in a program.
  - Then either producer may produce the data faster or consumer may retrieve an order of data and process it without its existing.

Thread



- Java uses the keyword **synchronized** to synchronize them and intercommunicate to each other.
- A mechanism which allows 2 or more threads to share all the available resources in a sequential manner.

Lock

- Lock term refers to the access granted to a particular thread that can access the shared resources.
- Java has build-in lock that only comes in action when the object has synchronized method code.
- No other thread can acquire the lock until the lock is not released by first thread.
- Acquire the lock means the thread currently in synchronized method and released the lock means exit the synchronized method.

Important Points

- Points for synchronization or lock:
  - Only methods or blocks can be synchronized
  - Each object has just only one lock
  - All methods in a class need not to be synchronized
  - If a thread goes to sleep, it hold any locks it has? It doesn't release them.
- Two ways to synchronize the execution of code
  - Synchronized Methods
  - Synchronized Blocks

Synchronization - Barrier

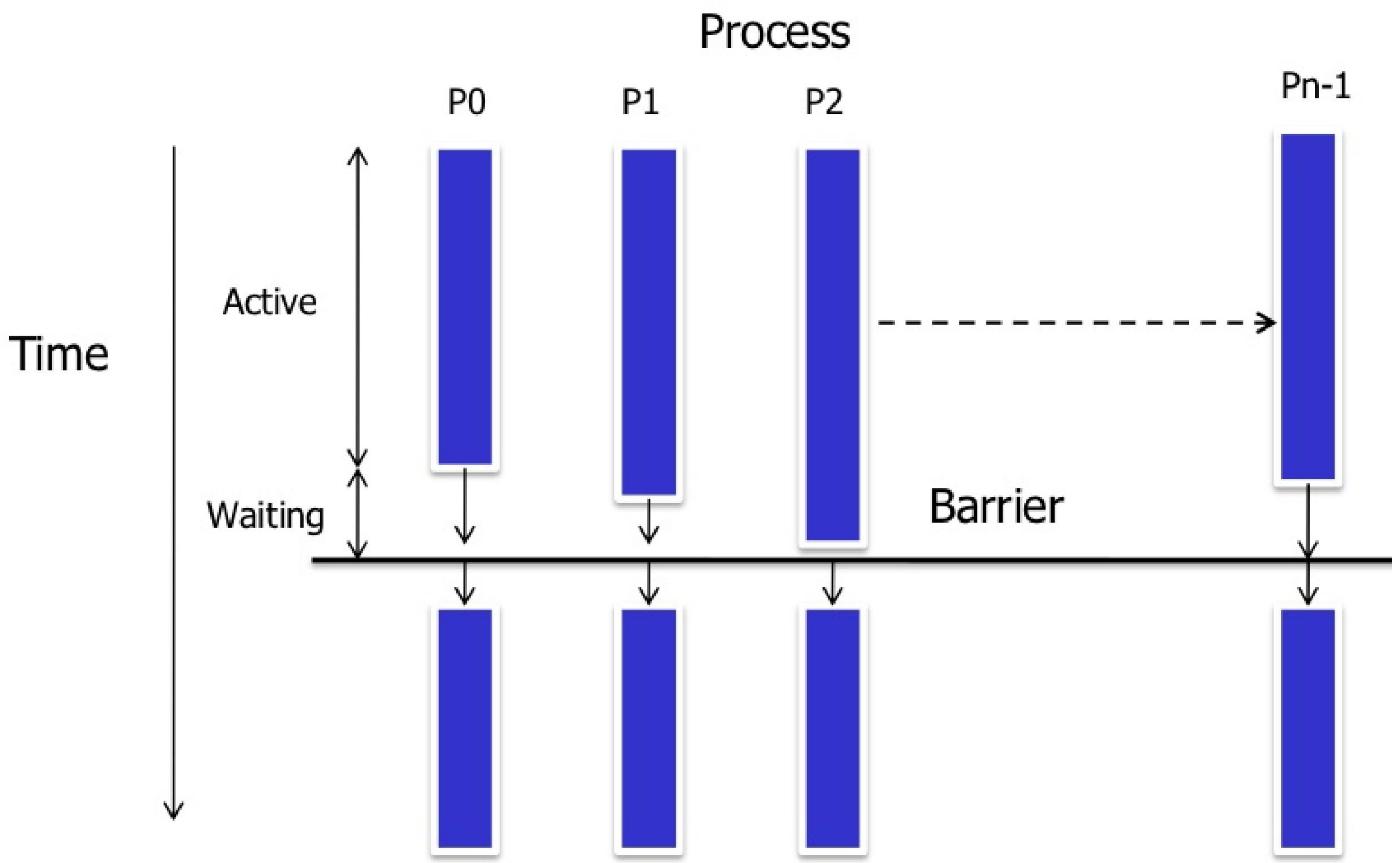
- We could start multiple threads each time around the loop, and wait for them all to complete
- This is inefficient, since we are continually spawning new processes
- This is much less efficient than having looping n processes and implementing synchronization

Synchronization - Barrier

- A barrier, a basic mechanism for synchronizing processes- inserted at the point in each process where it must wait
- All processes can continue from this point when all the processes have reached it
- In message-passing systems, barriers are often provided with library routines

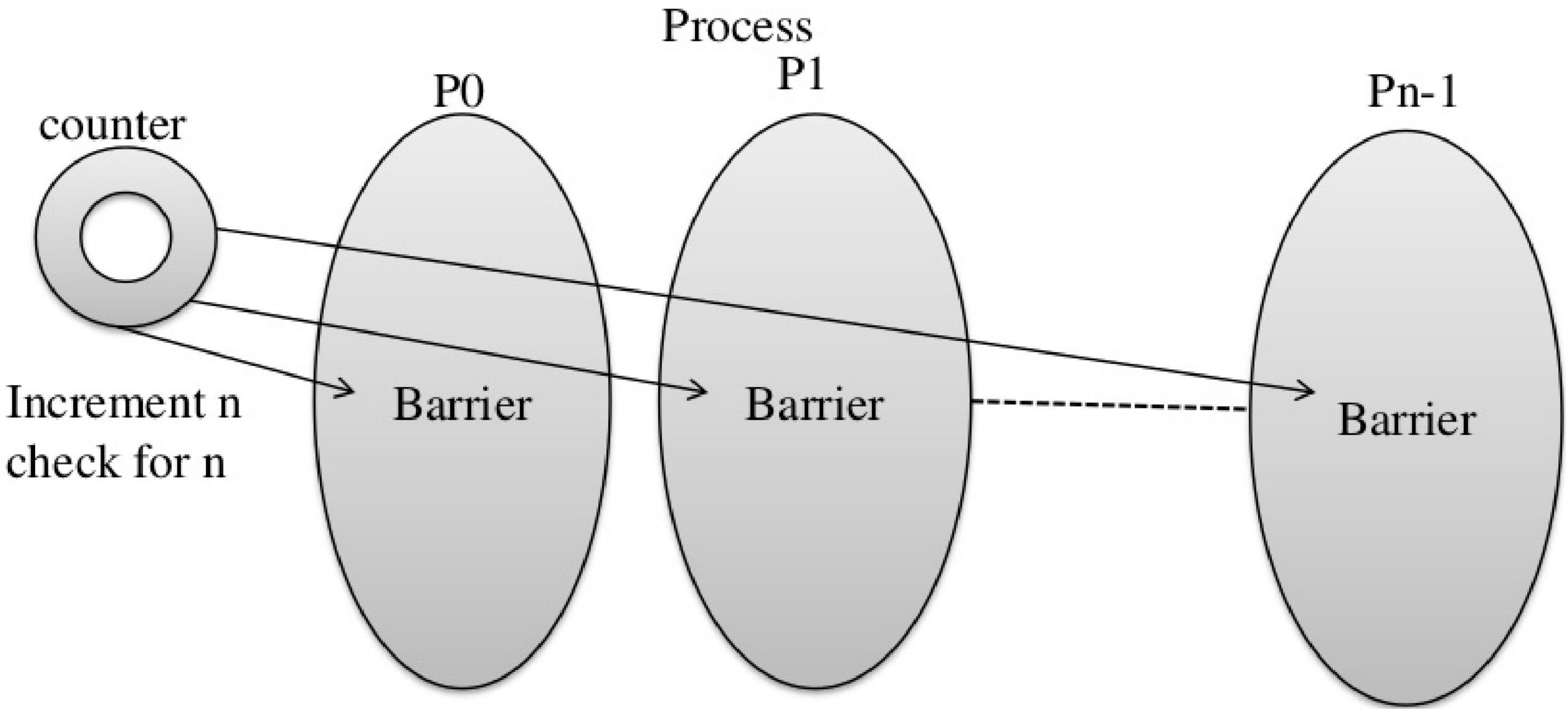
- MPI has the barrier routine, MPI_barrier()
- PVM has a similar barrier routine, pvm_barrier()

Barrier Example



Counter Implementation

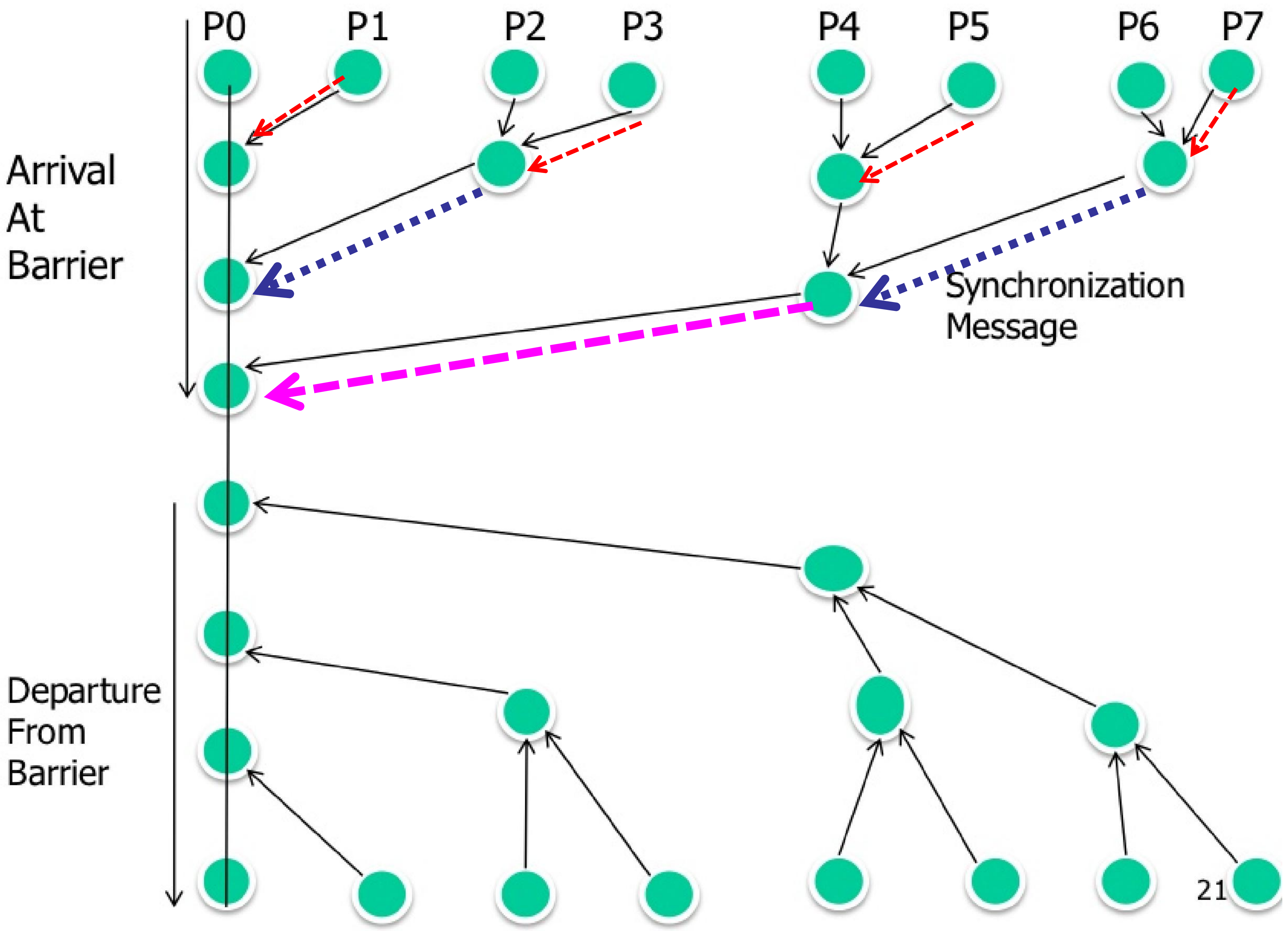
Centralized Counter implementation (sometimes called a linear barrier)



Tree Implementation

- More efficient. Suppose there are eight processes, P0, P1, P2, P3, P4, P5, P6 and P7:
- First Stage
  - P1 sends message to P0;
  - P3 sends message to P2;
  - P5 sends message to P4;
  - P7 sends message to P6;
- Second Stage
  - P2 sends message to P0;
  - P6 sends message to P4;
- Third Stage
  - P4 sends message to P0;
  - P0 terminates arrival phase

Tree Implementation

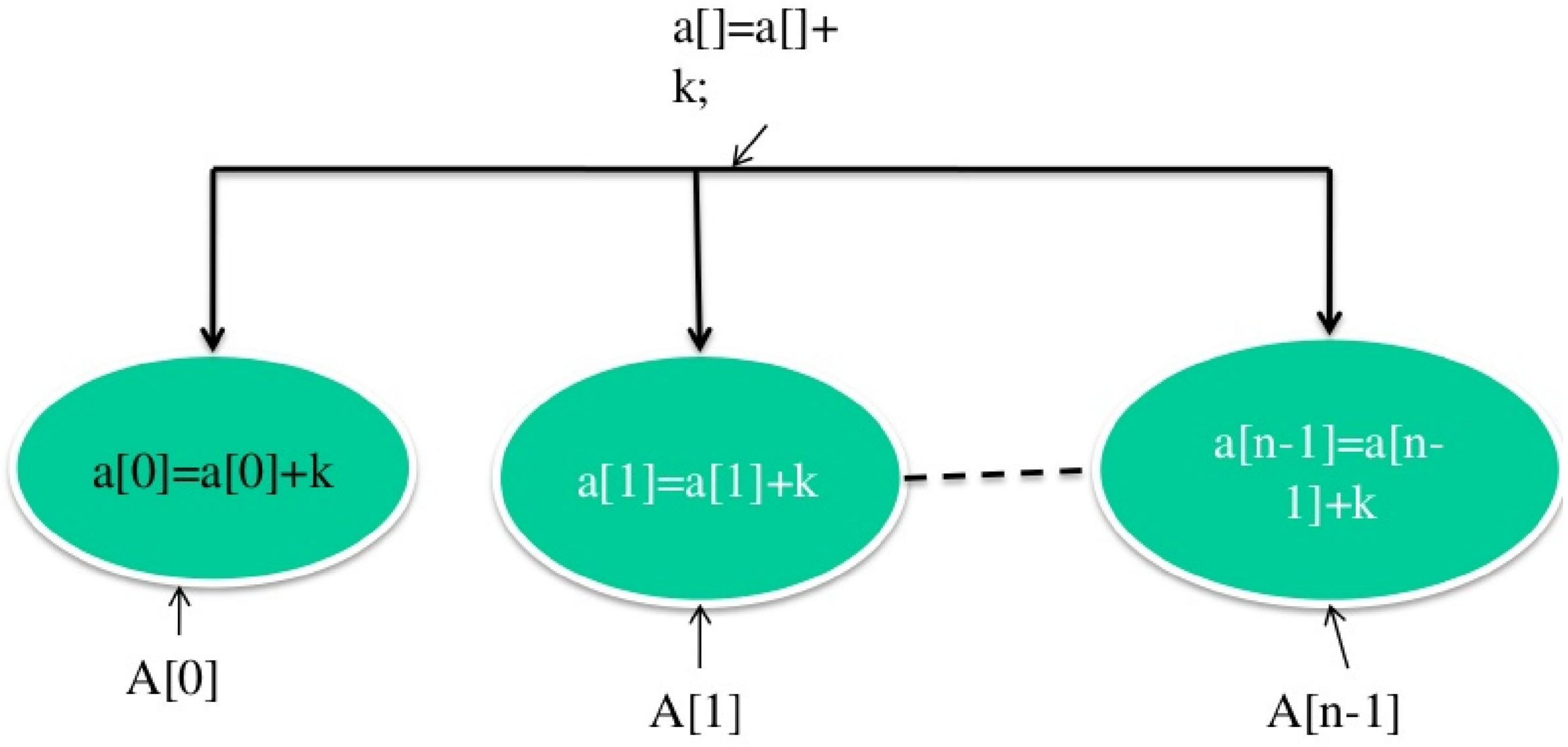


Data Parallel Computations

- Synchronization is required
- Same operation in different data element
- Data parallel programming is more convenient because:
  - Ease of programming
  - Scale easily to large problems
- Many numeric and non-numeric problem can be cast in data parallel form
- Example: SIMD computers
- SIMD Computers:
  - Same instruction executed on different processor but on different data type
  - Synchronization is built into the hardware

Example

- For(i=0;i<n;i++)
- a[i]=a[i]+k



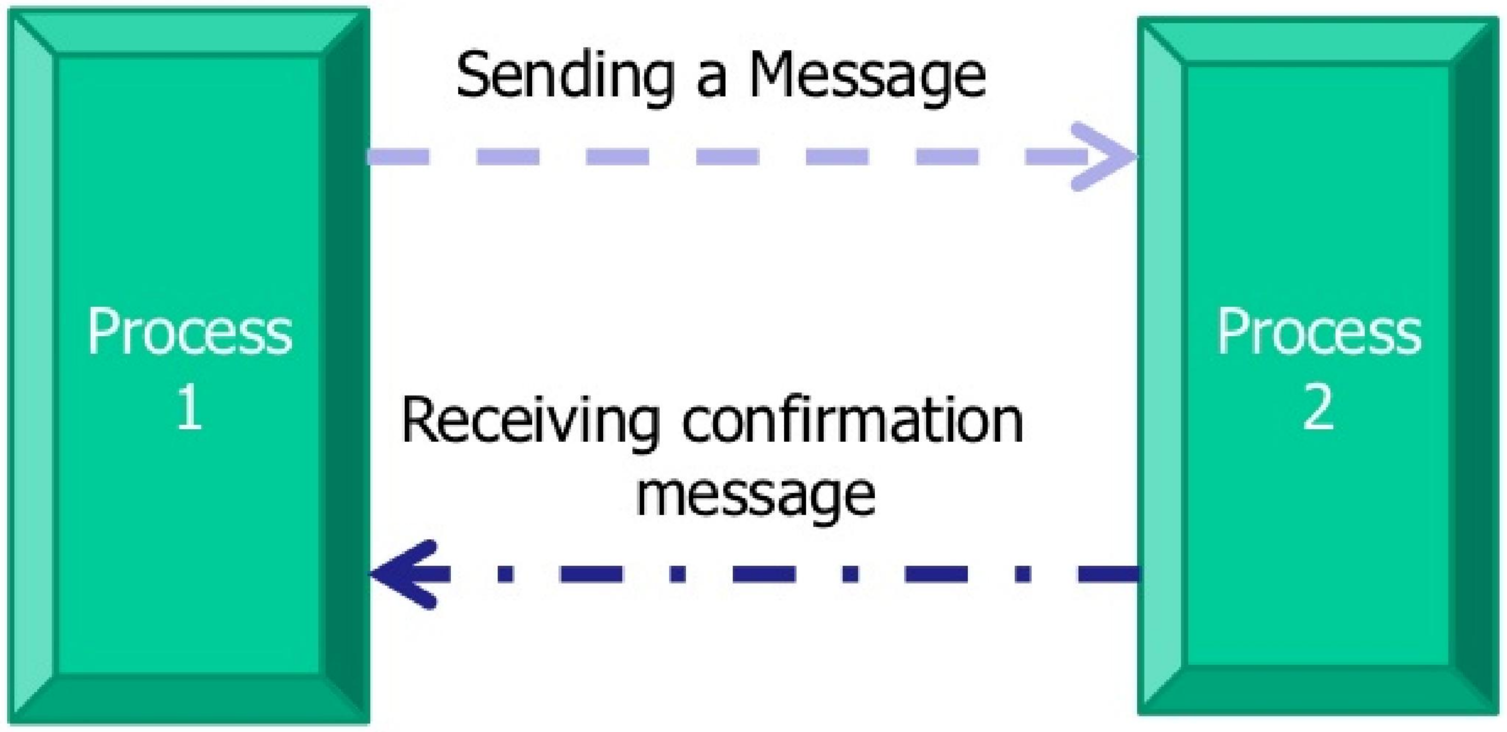
Barrier Requirement

- Data parallel technique is applied to Multiprocessor or Multicomputer
- The whole construct should not be completed before the instances thus a barrier is required

- Forall(i=0;i<n;i++)

```
        a[i]=a[i]+k
```

Butterfly Barrier

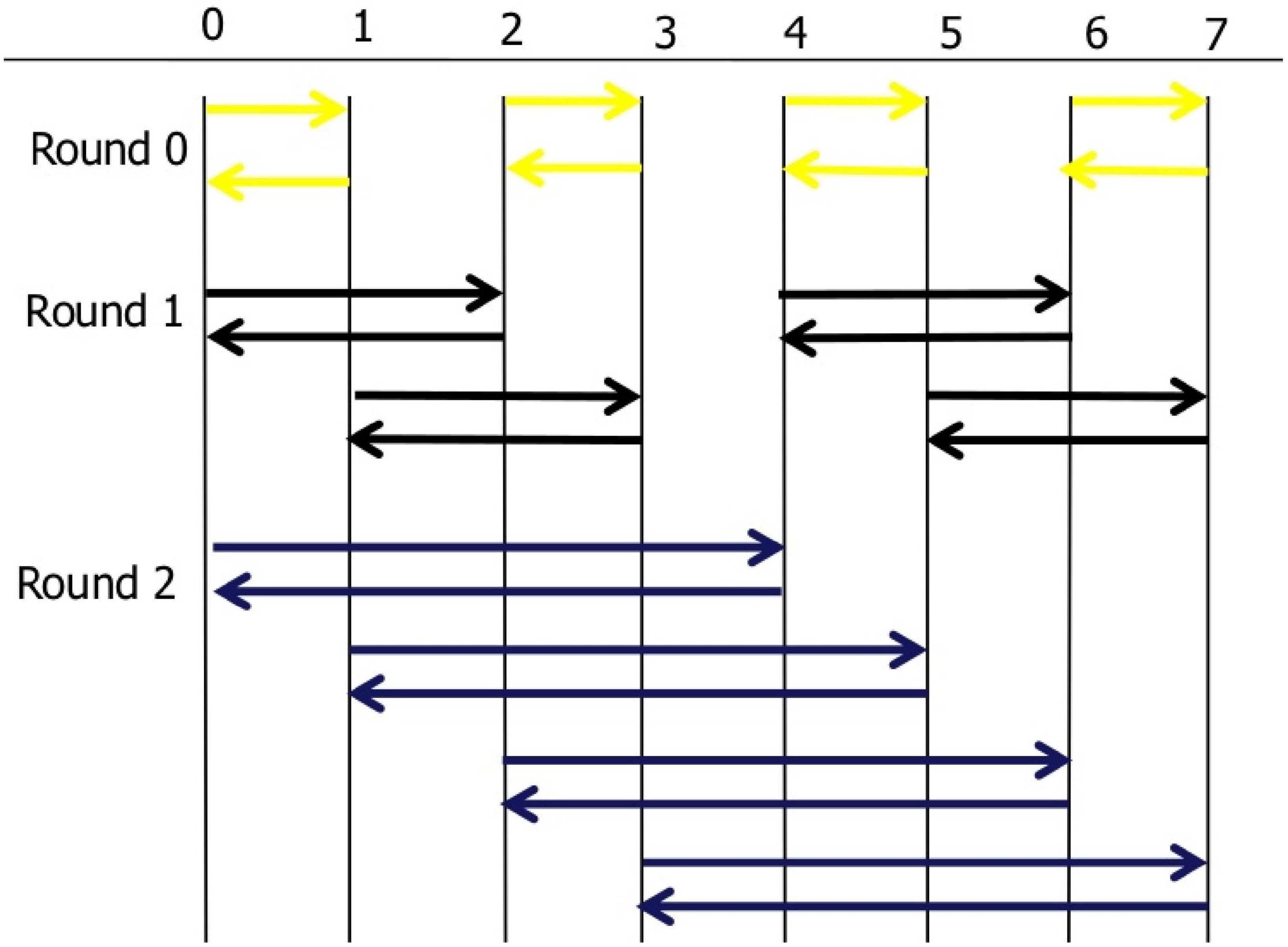


- Send a Message to partner process
- Wait until message is received from that process

Stages of Butterfly Barrier

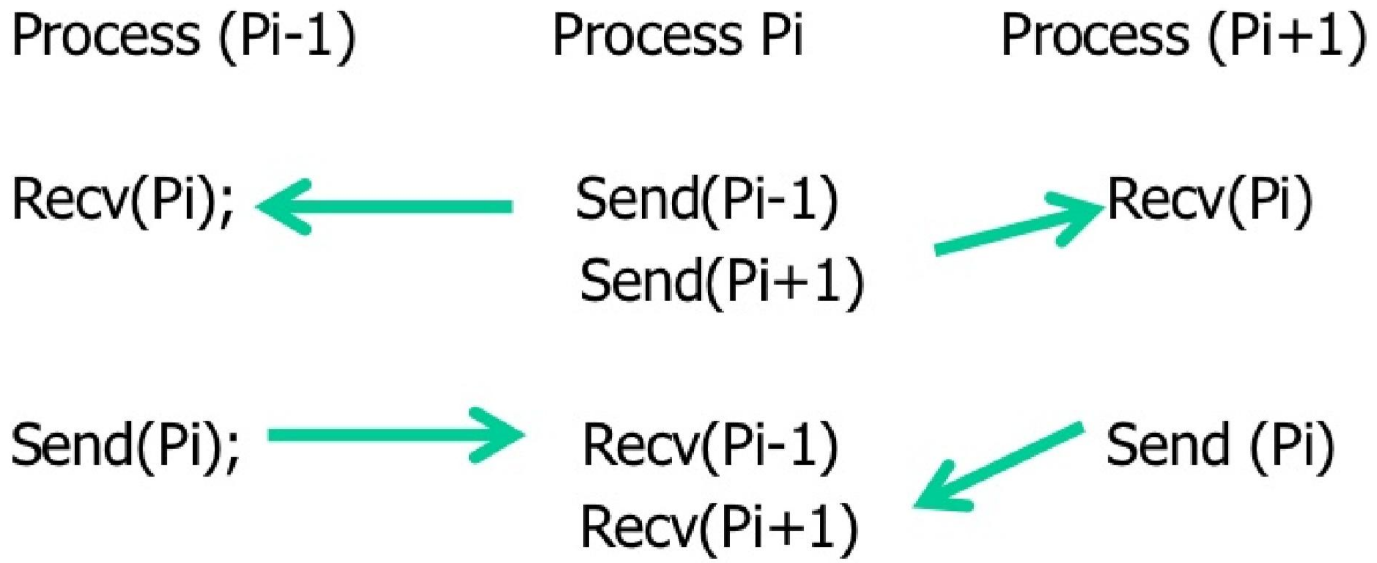
- If we have $n=2^k$ processes we build a barrier in k stages 1,2,…k
- At stage s processes synchronize with a partner that is 2s-1 steps away
- These are interleaved so that no process can pass through all stages in the barrier until all process have reached it
- If n isn't a power of 2 we can use the next largest 2k, but this isn't efficient and the system is no longer symmetric

Working Model of Butterfly Barrier



Local Synchronization

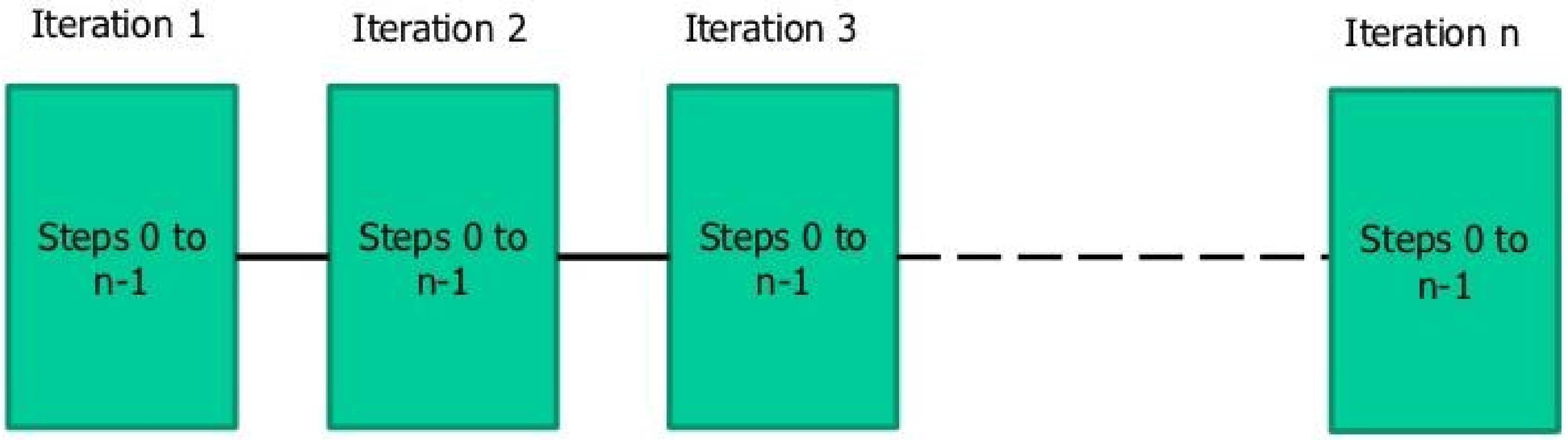
- Useful when calculation take varying amount of time and delays can occur randomly at any processor or task
- Create Batch



Note: Not a perfect three – process barrier Pi-1 will only synchronize with Pi and continue as soon as Pi allow

Synchronous Iteration (Synchronous Parallelism)

- Synchronous iteration: This term is used to describe a situation where a problem is solved by iteration and
- Each iteration step is composed of several processes that start together at the beginning of the iteration step and
- next iteration step cannot begin until all processes have finished the current iteration step



Iteration process diagram

Example 1: Synchronous Iteration

Equation: (4+6) –(2*3)

10-6=4

Squential solution : Solve above equation linear way and start with solve equation prioriy wise

10-6=4

Parallel solution : Solve above equation parallel way and from both side of tree

Example 2: Synchronous Iteration

- **Solving a General System of Linear Equations by Iteration**
- Suppose the equations are of a general form with n equations and n unknowns

$$
\begin{array}{lll}
a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \cdots & + a_{n-1,n-1}x_{n-1} & = b_{n-1} \\
\vdots & & \\
a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \cdots & + a_{2,n-1}x_{n-1} & = b_2 \\
a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \cdots & + a_{1,n-1}x_{n-1} & = b_1 \\
a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \cdots & + a_{0,n-1}x_{n-1} & = b_0
\end{array}
$$

$$x_i = \frac{1}{a_{i,i}}\left[b_i - \sum_{j \neq i} a_{i,j}x_j\right]$$

where the unknowns are $x_0, x_1, x_2, \ldots x_{n-1}$ ($0 <= i < n$).

Illustrating Jacobi Iteration

$7x_1 + 3x_2 + x_3 = 18$
$2x_1 - 9x_2 + 4x_3 = 12$
$x_1 - 4x_2 + 12x_3 = 6$

X11 =2.571 - 0.429(0) - 0.143(0) = 2.571
X12 = - 1.333+0.222 (0) +0.444 (0) = -1.333
X13 = 0.500 - 0.083 (0) + 0.333 (0) = 0.500

X1 = 18/7 - 3/7x2 - 1/7x3
X2 = -12/9 + 2/9x1 + 4/9x3
X3 = 6/12 - 1/12x1 + 4/12x2
Use as the initial estimates:
x1(0) = x2(0) = x3(0) = 0. Insert these estimates into these equations yielding new estimates of the parameters.

The estimated results after each iteration are shown as:

Iteration	x(1)	x(2)	x(3)
1	2.57143	- 1.33333	0.50000
2	3.07143	- 0.53968	- 0.15873
3	2.82540	- 0.72134	0.06415
4	2.87141	- 0.67695	0.02410
5	2.85811	- 0.68453	0.03506
6	2.85979	- 0.68261	0.03365

Sequential Code

```
for (i=0; i<n; i++)
  x[i] = b[i];
for (iter = 0; iter < limit; iter++)
{
   for (i=0; i<n; i++)
   {
       sum = 0;
       for (j=0; j<n; j++)
          if (i != j)
             sum = sum +  a[i][j]*x[j];
      newx[i] = (b[i]- sum)/a[i][i]

   }
   for (i=0; i<n; i++)
      x[i] = newx[i];
}
```

7x1 + 3x2 + x3 = 18
2x1 - 9x2 + 4x3 = 12
x1 - 4x2 + 12x3 = 6

Use as the initial estimates:
x1(0) = x2(0) = x3(0) = 0. Insert these estimates into these equations yielding new estimates of the parameters.

Iteration 1:
newx[0] = (18 – 0)/7 = 2.571

newx[1] = - (12 – 0)/9 = -1.333

newx[2] = (6 – 0)/12 = 0.500

x1(1) = 2.571 x2(1) = -1.333 x3(1) = 0.500

Iteration 2:
newx[0] = 2.571 +0.500357= 3.071

newx[1] = -1.333+0.792762 = - 0.540

newx[2] = 0.500 -0.657282 = - 0.158

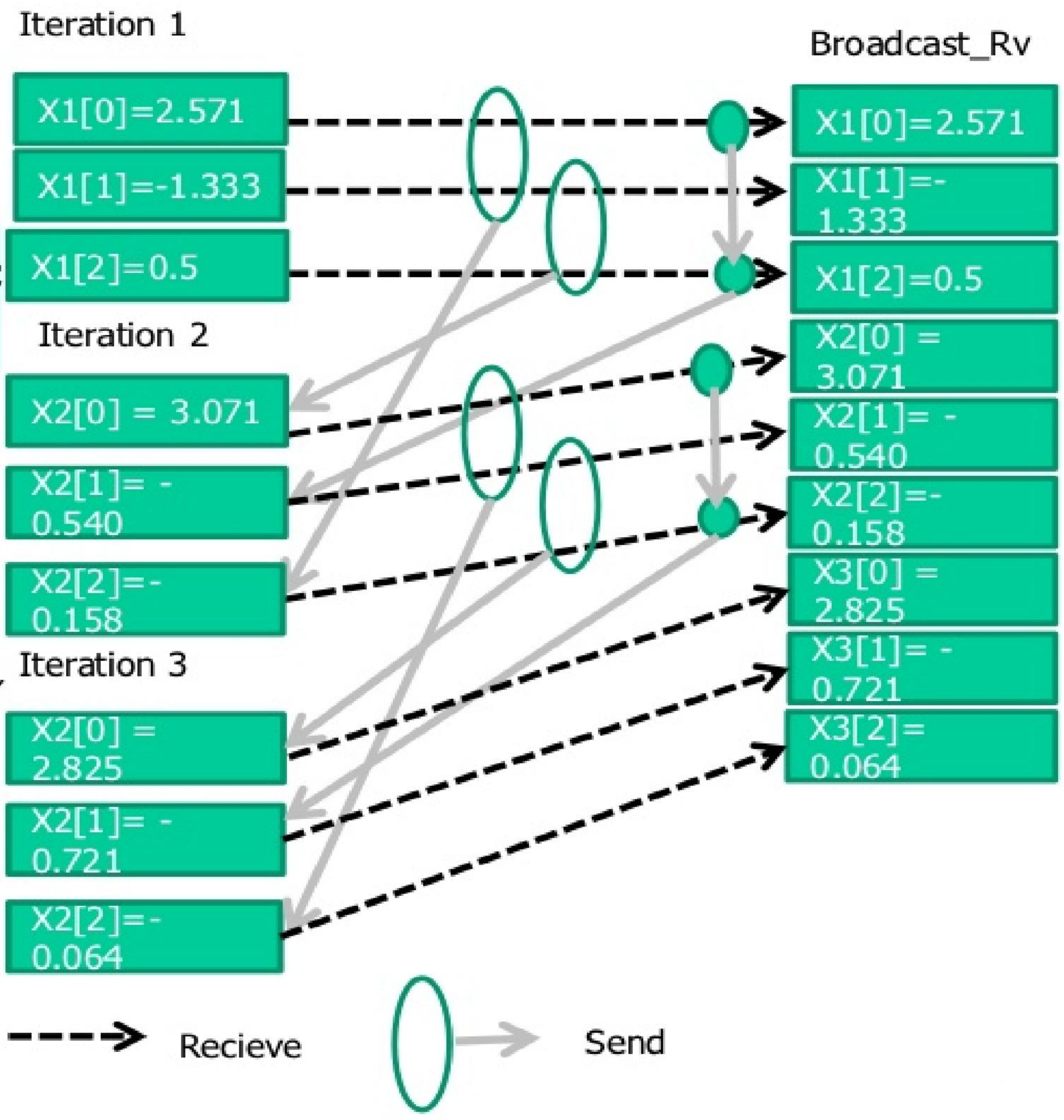
Parallel Code

- Suppose we have a process P_i for each unknown x_i; the code for process P_i may be:

```
 x[i] = b[i]
for (iter = 0; iter < limit; iter++)
{
   sum = -a[i][i] * x[i];
   for (j = 0; j < n; j++)
   sum = sum + a[i][j] * x[j];
   new_x[i] = (b[i] - sum) /a[i][i];
   broadcast_receive(&new_x[i]);
   global_barrier();
}
```

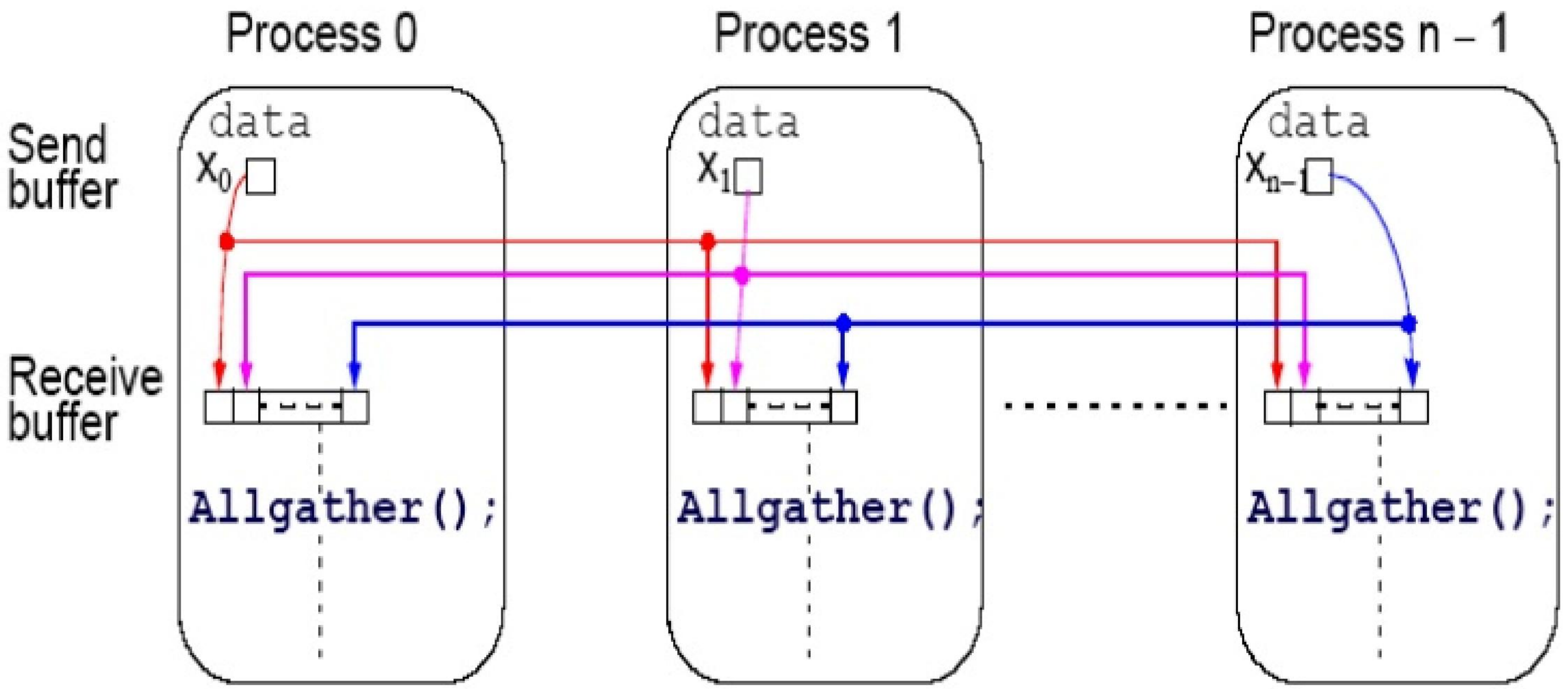
- broadcast receive() is used here

(1) to *send* the newly computed value of x[i] from process *Pi to every* other process and (2) to *collect* data broadcasted *from any* other processes to process *Pi.*



A New Message-Passing operation - Allgather

- Broadcast and gather values in one composite construction.



- *Note:* MPI Allgather() is also a global barrier, so you do not need to add global_barrier().

Solution by iteration

- Iterative Methods:
  - Applicable when direct methods require excessive computations
  - Have the advantage of small memory requirements
  - May not always converge / terminate
- An iterative method begins with an initial guess for the unknowns
  - Eg. $X_i = b_i$
- Iteration are continued until sufficiently accurate values obtained for the unknowns

DeadLock

References

1. http://www.slideshare.net/udaykumarsharma/parall el-programming-14005111

Process Synchronization Software

- Successful process synchronization in an OS requires that the OS:
  - Lock up a resource (e.g. printers, other I/O devices, memory locations, data files) in use by a process
    - Protect resource from other processes until released
  - Only when resource is released
    - Waiting process is allowed to use resource
- Mistakes in synchronization can result in:
  - Starvation
    - Leave job waiting indefinitely
  - Deadlock
    - If key resource is being used

Process Synchronization Software (continued)

- **Critical region**
  - Part of a program
  - A process must be allowed to finish work on a critical part of the program before other processes can have access to it. It is called a critical region because it is a critical section and its execution must be handled as a unit.
    - Other processes must wait before accessing critical region resources
- Processes within critical region
  - Cannot be interleaved
    - Threatens integrity of operation

Process Synchronization Software (continued)

- Synchronization
  - Implemented as lock-and-key arrangement:

    Step 1) Process determines key availability

    Step 2) If key is available, process picks up key, puts key in lock making it unavailable to other processes

    Both steps must be executed indivisibly for this scheme to work.
- Types of locking mechanisms
  - Test-and-set
  - WAIT and SIGNAL
  - Semaphores

Test-and-Set

- Indivisible machine instruction known as TS
- Executed in single machine cycle to see if key is available, and if it is, sets key to unavailable
- Actual key
  - Single bit in storage location: zero (free) or one (busy)
- Before process enters critical region
  - Tests condition code using TS instruction
  - No other process in region
    - Process proceeds
    - Condition code changed from zero to one
    - P1 exits: code reset to zero, allowing others to enter

Test-and-Set (continued)

- Advantages
  - Simple procedure to implement
  - Works well for small number of processes
- Drawbacks
  - Starvation
    - When many processes are waiting to enter a critical region, processes gain access in an arbitrary fashion
  - Busy waiting
    - Waiting processes remain in unproductive, resource-consuming wait loops

WAIT and SIGNAL

- Modification of test-and-set
  - Designed to remove busy waiting
- Two new mutually exclusive operations
  - WAIT and SIGNAL
  - Part of process scheduler’s operations
- WAIT
  - Activated when process encounters busy condition code
- SIGNAL
  - Activated when process exits critical region and condition code set to “free”

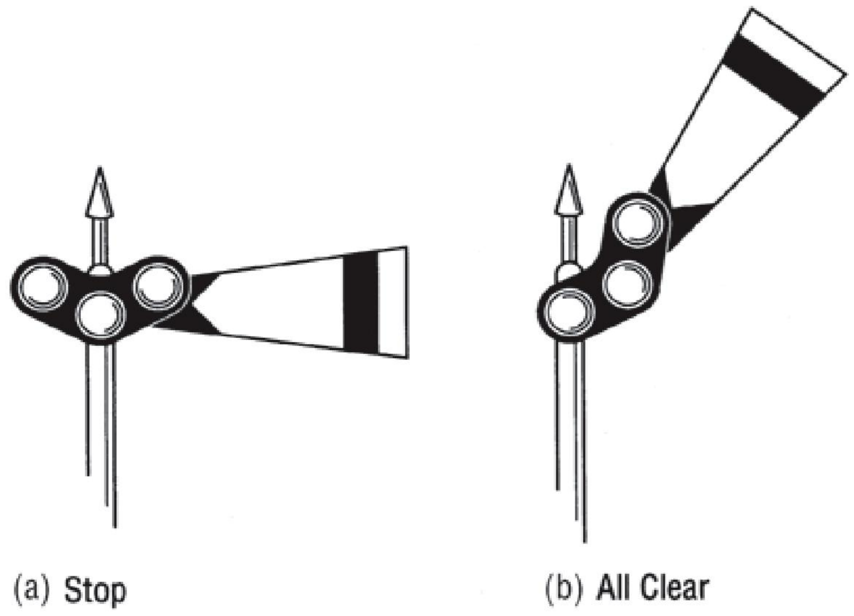
Semaphores

- Nonnegative integer variable
  - Is used as a flag
  - Signals if and when resource is free
    - Resource can be used by a process
- Two operations of semaphore
  - P (proberen means “to test”)
  - V (verhogen means “to increment”)

Semaphores (continued)

(figure 6.4)

The semaphore used by railroads indicates whether your train can proceed. When it's lowered (a), another train is approaching and your train must stop to wait for it to pass. If it is raised (b), your train can continue.



Semaphores (continued)

- Let *s* be a semaphore variable
  - V(*s*): *s*: = s + 1
    - Fetch, increment, store sequence
  - P(*s*): If *s* > 0, then *s*: = *s* – 1
    - Test, fetch, decrement, store sequence
- *s* = 0 implies busy critical region
  - Process calling on P operation must wait until *s* > 0
- Waiting job of choice processed next
  - Depends on process scheduler algorithm

Semaphores (continued)

State Number	Actions Calling Process	Operation	Running in Critical Region	Results Blocked on *s*	Value of *s*
0					1
1	P1	test(*s*)	P1		0
2	P1	increment(*s*)			1
3	P2	test(*s*)	P2		0
4	P3	test(*s*)	P2	P3	0
5	P4	test(*s*)	P2	P3, P4	0
6	P2	increment(*s*)	P3	P4	0
7			P3	P4	0
8	P3	increment(*s*)	P4		0
9	P4	increment(*s*)			1

(table 6.3)

The sequence of states for four processes calling test and increment (P and V) operations on the binary semaphore s. (Note: The value of the semaphore before the operation is shown on the line preceding the operation. The current value is on the same line.)

Semaphores (continued)

- P and V operations on semaphore *s*
  - Enforce mutual exclusion concept necessary to avoid having two operations attempt to execute at the same time.
- Semaphore traditionally called **mutex** (MUTual EXclusion)

    P(mutex): if mutex > 0 then mutex: = mutex – 1

    V(mutex): mutex: = mutex + 1
- **Critical region**
  - Ensures parallel processes modify shared data only while in critical region

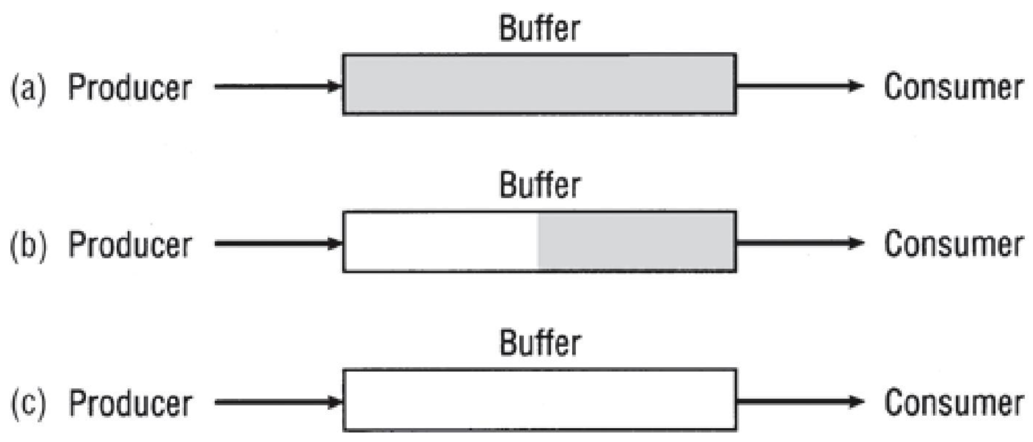
Process Cooperation

- Several processes work together to complete common task
- Each case of process cooperation requires
  - Mutual exclusion and synchronization
- Absence of mutual exclusion and synchronization results in problems
- Example
  - Producers and consumers problem
  - implemented using semaphores

Producers and Consumers

- One process produces data
- Another process later consumes data
- Example: CPU and printer buffer. Because the buffer is finite, the synchronization process must:
  - delay the producer from generating more data when buffer is full
  - Must also delay the consumer from retrieving data when buffer is empty
  - Implemented by two counting semaphores
    - Number of full positions
    - Number of empty positions
  - Mutex
    - Third semaphore: ensures mutual exclusion

Producers and Consumers (continued)



(figure 6.5)

The buffer can be in any one of these three states: (a) full buffer, (b) partially empty buffer, or (c) empty buffer.

Producers and Consumers (continued)

Producer	Consumer
produce data	P (full)
P (empty)	P (mutex)
P (mutex)	read data from buffer
write data into buffer	V (mutex)
V (mutex)	V (empty)
V (full)	consume data

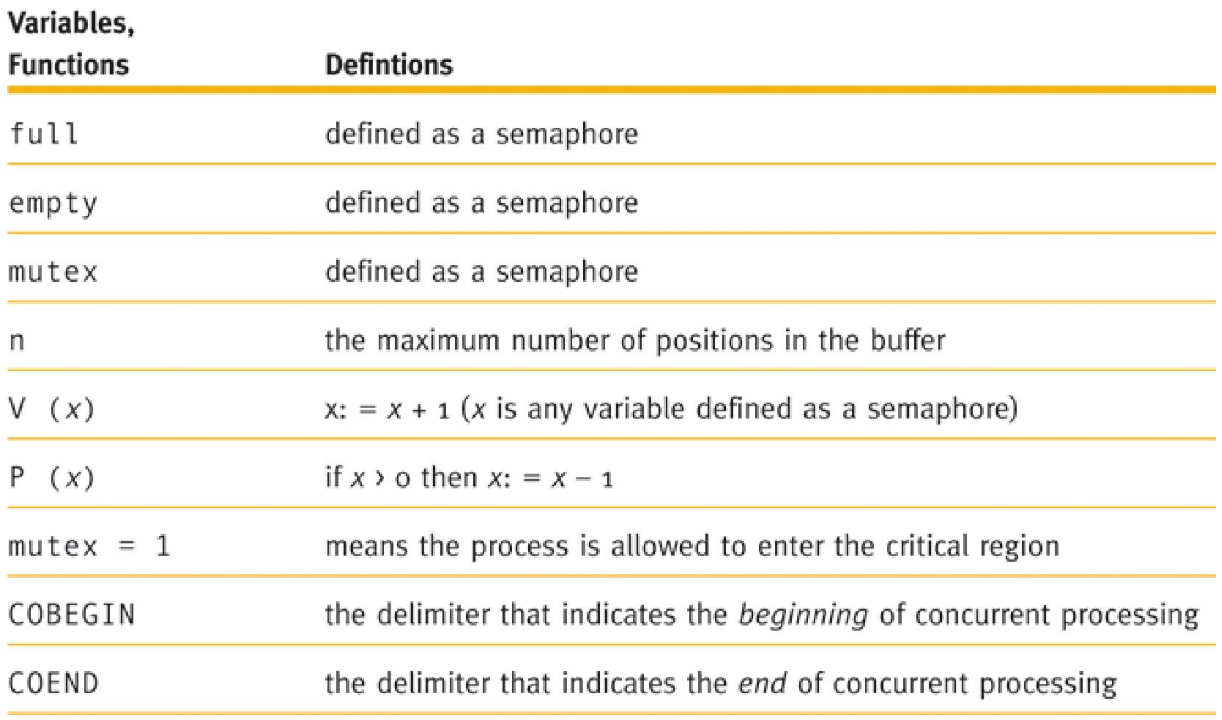
(table 6.4)

Definitions of the Producers and Consumers processes.

Producers and Consumers (continued)

(table 6.5)

Definitions of the elements in the Producers and Consumers Algorithm.



Variables, Functions	Defintions
`full`	defined as a semaphore
`empty`	defined as a semaphore
`mutex`	defined as a semaphore
`n`	the maximum number of positions in the buffer
`V (x)`	$x: = x + 1$ (x is any variable defined as a semaphore)
`P (x)`	if $x > 0$ then $x: = x - 1$
`mutex = 1`	means the process is allowed to enter the critical region
`COBEGIN`	the delimiter that indicates the *beginning* of concurrent processing
`COEND`	the delimiter that indicates the *end* of concurrent processing

Producers and Consumers (continued)

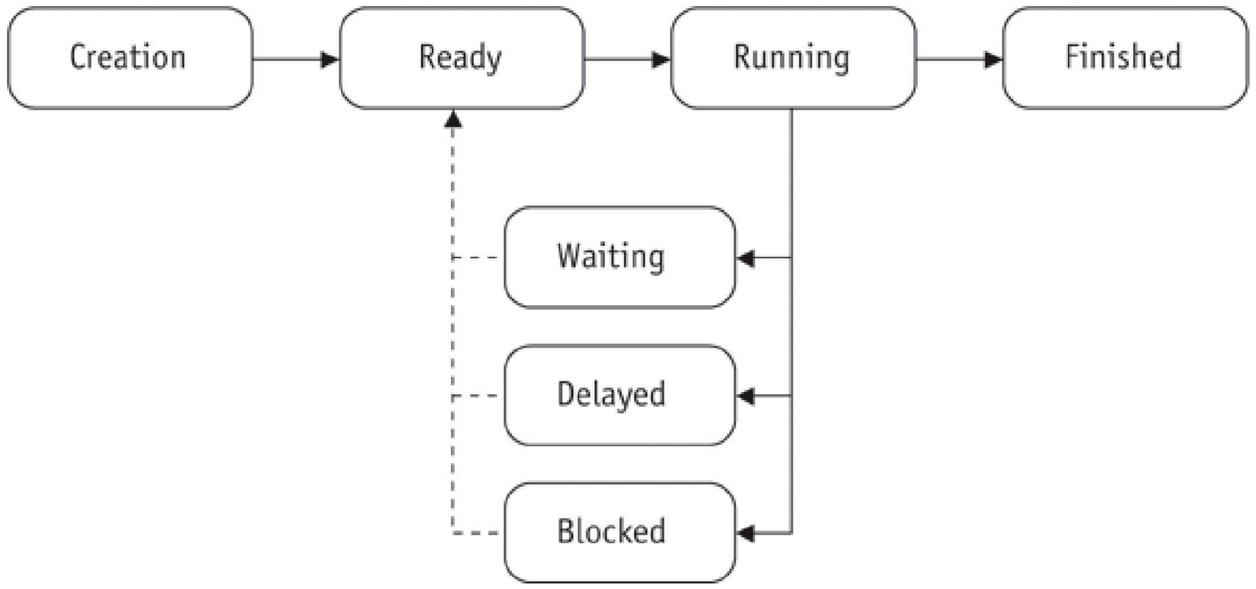
- Producers and Consumers Algorithm

```
empty: = n
full: = 0
mutex: = 1
COBEGIN
  repeat until no more data PRODUCER
  repeat until buffer is empty CONSUMER
COEND
```

Threads and Concurrent Programming

- **Threads**
  - Small unit within process
    - Scheduled and executed
- Minimizes overhead of swapping process between main memory and secondary storage
- Each active process thread
  - Processor registers, program counter, stack and status
- Shares data area and resources allocated to its process

Thread States



(figure 6.6)

A typical thread changes states from READY to FINISHED as it moves through the system.

Thread States (continued)

- Operating system support
  - Creating new threads
  - Setting up thread
    - Ready to execute
  - Delaying or putting threads to sleep
    - Specified amount of time
  - Blocking or suspending threads
    - Those waiting for I/O completion
  - Setting threads to WAIT state
    - Until specific event occurs

Thread States (continued)

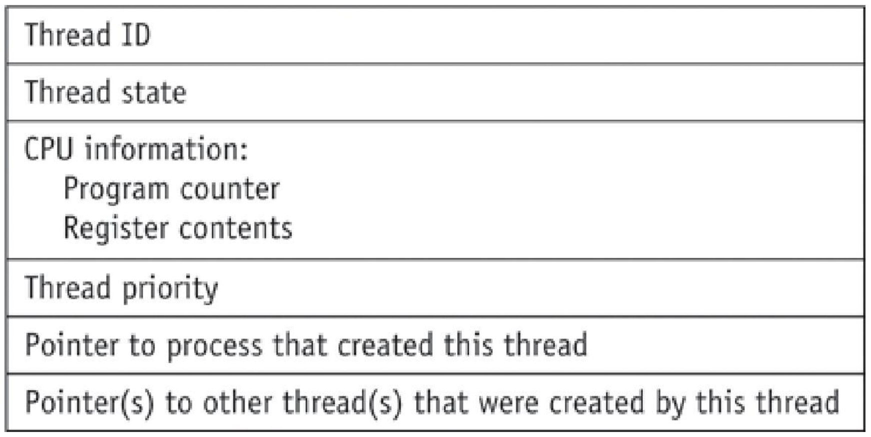
- Operating system support (continued)
  - Scheduling thread execution
  - Synchronizing thread execution
    - Using semaphores, events, or conditional variables
  - Terminating thread
    - Releasing its resources

Thread Control Block

- Information about current status and characteristics of thread

(figure 6.7)

The contents of a typical Thread Control Block (TCB).



Thread ID
Thread state
CPU information: Program counter Register contents
Thread priority
Pointer to process that created this thread
Pointer(s) to other thread(s) that were created by this thread

Concurrent Programming Languages

- **Java**
  - Designed as universal Internet application software platform
  - Developed by Sun Microsystems
  - Adopted in commercial and educational environments

Java

- Allows programmers to code applications that can run on any computer
- Developed at Sun Microsystems, Inc. (1995)
- Solves several issues
  - High software development costs for different incompatible computer architectures
  - Distributed client-server environment needs
  - Internet and World Wide Web growth
- Uses compiler and interpreter
  - Easy to distribute

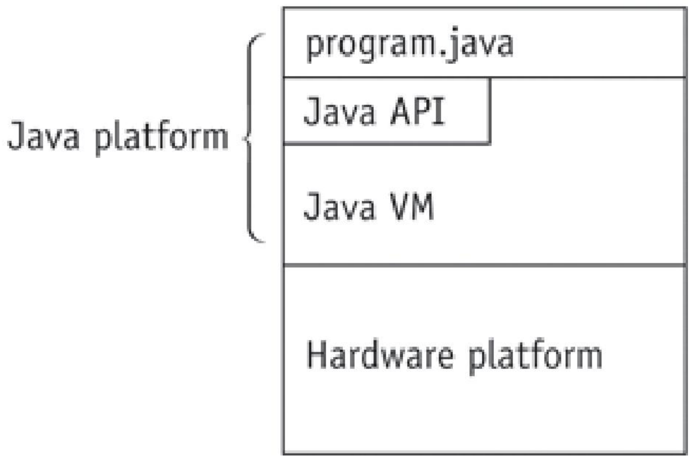
Java (continued)

- **The Java Platform**
- Software only platform
  - Runs on top of other hardware-based platforms
- Two components
  - Java Virtual Machine (Java VM)
    - Foundation for Java platform
    - Contains the interpreter
    - Runs compiled bytecodes
  - Java application programming interface (Java API)
    - Collection of software modules
    - Grouped into libraries by classes and interfaces

Java (continued)

(figure 6.8)

A process used by the Java platform to shield a Java program from a computer's hardware.



Java (continued)

- **The Java Language Environment**
- Designed for experienced programmers (like C++)
- Object oriented
  - Exploits modern software development methods
    - Fits into distributed client-server applications
- Memory allocation features
  - Done at run time
  - References memory via symbolic “handles”
  - Translated to real memory addresses at run time
  - Not visible to programmers

Java (continued)

- Security
  - Built-in feature
    - Language and run-time system
  - Checking
    - Compile-time and run-time
- Sophisticated synchronization capabilities
  - Multithreading at language level
- Popular features
  - Handles many applications; can write a program once; robust; Internet and Web integration