Concurrency: Mutual Exclusion and Synchronization

Needs of Processes

- Allocation of processor time
- Allocation and sharing resources
- Communication among processes
- Synchronization of multiple processes

Have seen: scheduling, memory allocation

Now: synchronization (incl. communication)

Next: more on resource allocation & deadlock

Shared memory communication

- Ex1: memory mapped files (cf memory)
- Ex2: POSIX Shared Memory
  - Process first creates shared memory segment

```
segment id = shmget(key, size, S IRUSR | S IWUSR);
```

  - Process wanting access to that shared memory must attach to it

```
segment id = shmget(key, …, …)
shared memory = (char *) shmat(id, NULL, 0);
```

  - Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared memory");
```

  - When done a process can detach the shared memory from its address space

```
shmdt(shared memory);
```

Check
www.cs.cf.ac.uk/Dave/C/node27.html#SECTION002720000000000000

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, sockets)
  - logical (e.g., logical properties)

Message Passing Systems

- Interaction =
  - synchronization (mutex, serializations, dependencies,…)
  - communication (exchange info)
- message-passing can do both simultaneously: Send/receive can be blocking or non-blocking
  - both blocking : rendez-vous
  - can also have interrupt-driven receive
- source, destination can be process or mailbox/port

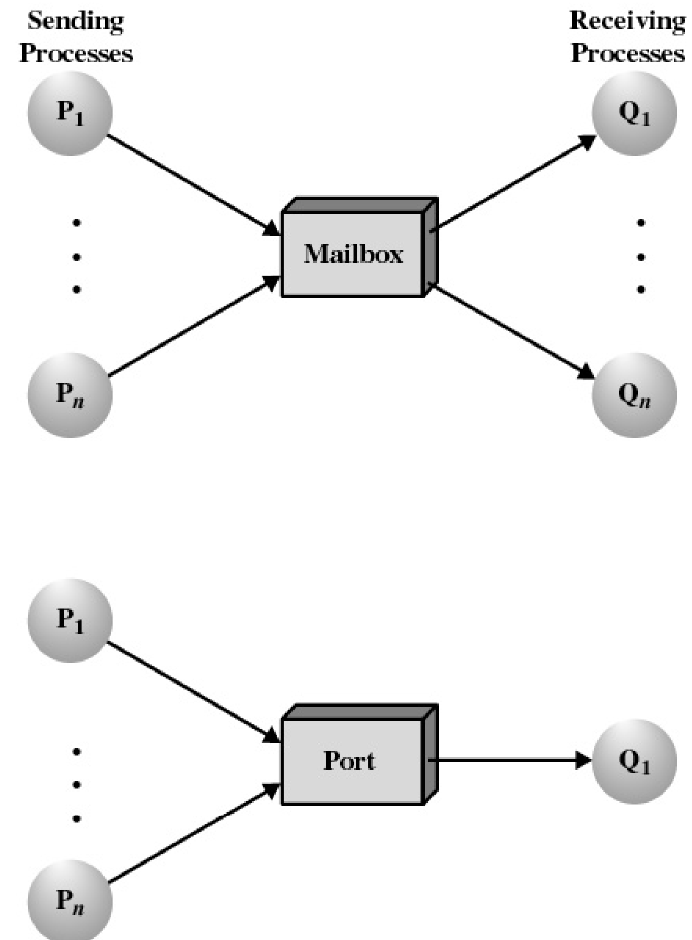


Figure 5.24 Indirect Process Communication

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Money flies away ...

BALANCE: 20000 kr

Bank thread A	Bank thread B
Read a := BALANCE	
a := a + 5000	Read b:= BALANCE
Write BALANCE := a	b := b – 2000
	Write BALANCE := b

Oooops!! BALANCE: 18000 kr!!!!

Problem: need to ensure that each process is executing its critical section (e.g updating BALANCE) exclusively (one at a time)

Process Synchronization: Roadmap

- The critical-section (mutual exclusion) problem
- Synchronization for 2 and for n processes using shared memory
- Synchronization hardware
- Semaphores

- Other common synchronization structures
- Common synchronization problems

- Synchronization in message passing systems

The Critical-Section (Mutual Exclusion) Problem

- n processes all 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 process is allowed to execute in its critical section; ie. Access to the critical section must be an **atomic action**.
- Structure of process P_i

repeat
    entry section

        critical section
    exit section

        remainder section
until *false*;

Requirements from a solution to the Critical-Section Problem

1. **Mutual Exclusion**. Only one process at a time is allowed to execute in its critical section.

2. **Progress (no deadlock)**. If no process is executing in its critical section and there exist some processes that wish to enter theirs, the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. **Bounded Waiting (no starvation)**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed and that each process remains in its critical section for finite time only
   - No assumption concerning relative speed of the *n* processes.

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- Processes may share some common variables to synchronize their actions.

Shared variables:

- **var** *turn*: (0..1) (initially 0)
- $turn = i \Rightarrow P_i$ can enter its critical section

Process P_i

repeat

while $turn \neq i$ **do** *no-op*;

critical section

$turn := j$;

remainder section

until *false*;

- (too polite) Satisfies mutual exclusion, but not progress

Another attempt

Shared variables

- **var** *flag*: **array** [0..1] **of** *boolean*; (initially *false*).
- *flag* [i] = *true* $\Rightarrow P_i$ ready to enter its critical section

Process P_i

repeat

while *flag*[*j*] **do** *no-op*;

flag[*i*] := *true*;

critical section

flag [*i*] := *false*;

remainder section

until *false*;.

- (unpolite) Progress is ok, but does NOT satisfy mutual exclusion.

Peterson's Algorithm (2 processes)

Shared variables:

var *turn*: (0..1); initially 0 (*turn* = *i* ⇒ P_i can enter its critical section)

var *flag*: **array** [0..1] **of** *boolean*; initially *false* (*flag* [*i*] = *true* ⇒ P_i wants to enter its critical section)

Process P_i

```
repeat
    (F) flag [i] := true;
    (T) turn := j;
    (C) while (flag [j] and turn = j) do no-op;
        critical section
    (E) flag [i] := false;
        remainder section
until false;
```

Prove that it satisfies the 3 requirements

Mutual Exclusion: Hardware Support

- Interrupt Disabling
  - A process runs until it invokes an operating-system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion

    - BUT:
      - Processor is limited in its ability to interleave programs
      - Multiprocessors: disabling interrupts on one processor will not guarantee mutual exclusion

Mutual Exclusion: Other Hardware Support

- Special Machine Instructions
  - Performed in a single instruction cycle: Reading and writing together as one atomic step
  - Not subject to interference from other instructions
    - in uniprocessor system they are executed without interrupt;
    - in multiprocessor system they are executed with locked system bus

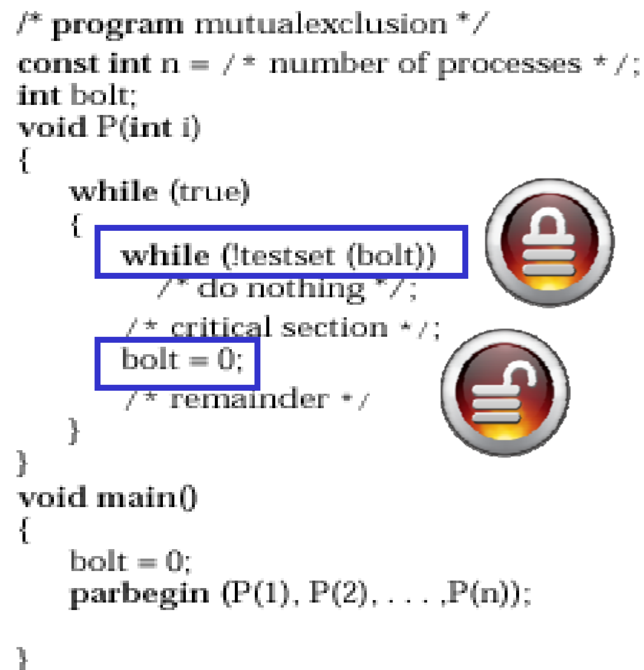
Mutual Exclusion: Hardware Support

Test and Set Instruction

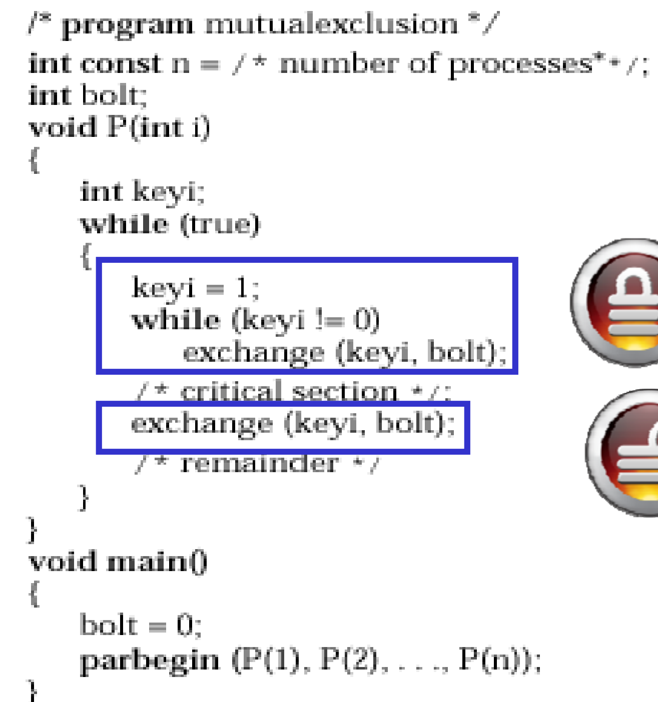
boolean testset (int i)
if (i == 0)
i = 1; return true;
else return false;

Exchange Instruction (swap)

void exchange(int mem1, mem2)
temp = mem1;
mem1 = mem2;
mem2 = temp;



```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true)
    {
        while (!testset (bolt))
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . ,P(n));
}
```



```
/* program mutualexclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
    int keyi;
    while (true)
    {
        keyi = 1;
        while (keyi != 0)
            exchange (keyi, bolt);
        /* critical section */;
        exchange (keyi, bolt);
        /* remainder */
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . ., P(n));
}
```

Mutual Exclusion using Machine Instructions

Advantages

- Applicable to any number of processes on single or multiple processors sharing main memory
- It is simple and therefore easy to verify

Disadvantages

- Busy-waiting consumes processor time
- Starvation is possible when using such a simple methods; cf next for maintaining turn
- Deadlock possible if used in priority-based scheduling systems: ex. scenario:
  - low priority process has the critical region
  - higher priority process needs it
  - the higher priority process will obtain the processor to wait for the critical region

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;
                // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])  // find next one waiting and signal //
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
                // remainder section
} while (TRUE);
```

Semaphores

- Special variables used for signaling
  - If a process is waiting for a signal, it is blocked until that signal is sent
- Accessible via *atomic Wait* and *signal* operations
- Queue is (can be) used to hold processes waiting on the semaphore
- Can be binary or general (counting)

Binary and Counting semaphores

```
struct binary_semaphore {
        enum (zero, one) value;
        queueType queue;
};

void waitB(binary_semaphore s)
{
    if (s.value ==  1)
            s.value = 0;
    else
        {
            place this process in s.queue;
            block this process;
        }
}
void signalB(semaphore s)
{
    if (s.queue.is_empty())
            s.value = 1;
    else
        {
            remove a process P from s.queue;
            place process P on ready list;
        }
}
```

```
struct semaphore {
        int count;
        queueType queue;
}

void wait(semaphore s)
{
        s.count--;
        if (s.count < 0)
        {
            place this process in s.queue;
            block this process
        }
}
void signal(semaphore s)
{
        s.count++;
        if (s.count <= 0)
        {
            remove a process P from s.queue;
            place process P on ready list;
        }
}
```

Example: Critical section of *n* processes using semaphores

- Shared variables
  - **var** *mutex* : *semaphore*
  - initially *mutex* = 1
- Process P_i

repeat

    wait(*mutex*);

        critical section

    signal(*mutex*);

        remainder section

until *false*;

Semaphore as General Synchronization Tool

- E.g. execute *B* in P_j only after *A* executed in P_i: use semaphore *flag* initialized to 0

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

Watch for Deadlocks!!!

Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

More on synchronization

- Synchronization using semaphores, implementing counting semaphores from binary ones, etc
- Other high-level synchronization constructs
  - (conditional) critical regions (wait-until-value, …)
  - monitors
- Classical Problems of Synchronization
  - Bounded-Buffer (producer-consumer)
  - Readers and Writers (later, with lock-free synch)
  - Dining-Philosophers (Resource allocation: later, with deadlock avoidance)

train on these: it is very useful and fun!

Bounded Buffer

- *N* locations, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

producer process

```
do {
        //   produce an item

        wait (empty);
        wait (mutex);

        //  add the item to the  buffer

         signal (mutex);
         signal (full);

   } while (TRUE);
```

consumer process

```
do {
        wait (full)
        wait (mutex);

       //  remove an item from buffer

        signal (mutex);
        signal (empty);

         //  consume the item

   } while (TRUE);
```

Lamport's Bakery Algorithm (Mutex for n processes)

Idea:

- Before entering its critical section, each process receives a number. Holder of the smallest number enters the critical section.
  - If processes P_i and P_j receive the same number: if $i < j$, then P_i is served first; else P_j is served first.
- numbering scheme generates numbers in non-decreasing order of enumeration; i.e., 1,2,3,3,3,3,4,5
- *A distributed algo: uses no variable "writ-able" by all processes*

Lamport's Bakery Algorithm (cont)

```
Shared var choosing: array [0..n – 1] of boolean (init false);
           number: array [0..n – 1] of integer (init 0),
repeat
    choosing[i] := true;
    number[i] := max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] := false;
    for j := 0 to n – 1 do begin
        while choosing[j] do no-op;
        while number[j] ≠ 0 and (number[j],j) < (number[i], i) do
              no-op;
    end;
          critical section
    number[i] := 0;
          remainder section
until false;
```

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as mutexes and semaphores
  - Dispatcher objects may also provide events (like a condition variable)

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables

- Non-portable extensions include:
  - read-write locks
  - spin locks

Mutual exclusion using messages: Centralized Approach

Key idea: One processes in the system is chosen to *coordinate* the entry to the critical section (CS):

- A process that wants to enter its CS sends a *request* message to the coordinator.
- The coordinator decides which process can enter its CS next, and sends to it a *reply* message
- After exiting its CS, that process sends a *release* message to the coordinator

Requires 3 messages per critical-section entry (request, reply, release)

Depends on the coordinator (bottleneck)

Mutual exclusion using message-box: (pseudo) decentralized approach

Key idea: use a token that can be left-at/removed-from a common mailbox

Requires 2 messages per critical-section entry (receive-, send-token)

Depends on a central mailbox (bottleneck)

```
/* program mutualexclusion */
const int n = /* number of processes */;
void P(int i)
{
    message msg;
    while (true)
    {
        receive (mutex, msg);
        /* critical section */;
        send (mutex, msg);
        /* remainder */;
    }
}
void main()
{
    create_mailbox (mutex);
    send (mutex, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Producer(s)-consumer(s) (bounded-buffer) using messages

Key idea: similar as in the previous mutex solution:

- use producer-tokens to allow produce actions (to non-full buffer)
- use consume-tokens to allow consume-actions (from non-empty buffer)

```
const int
    capacity =  /* buffering capacity */ ;
    null =     /* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true)
    {
        receive (mayproduce, pmsg);
        pmsg = produce();
        send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true)
    {
        receive (mayconsume, cmsg);
        consume (cmsg);
        send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++)
        send (mayproduce, null);
    parbegin (producer, consumer);
}
```

Distributed Algorithm

- Each node has only a partial picture of the total system and must make decisions based on this information
- All nodes bear equal responsibility for the final decision
- There exits no system-wide common clock with which to regulate the time of events

Mutual exclusion using messages: distributed approach using token-passing

Key idea: use a token (message *mutex*) that circulates among processes in a *logical ring*

Process P_i

repeat

receive(P_{i-1}, *mutex*);

critical section

send(P_{i+1}, *mutex*);

remainder section

until *false*;

(if *mutex is received when not-needed, must be passed to* P_{i+1} at once)

Requires 2 (++) messages; can optimize to pass the token around on-request

Mutex using messages: fully distributed approach based on event ordering

Key idea: similar to bakery algo (relatively order processes' requests) [Rikard&Agrawala81]

Process *i*

```
when state_i=requesting
   state_i := wait;
   oks := 0;
   req_num_i := ++C_i;
   forall k send(k, req, req_num_i)
when receive(k,ack)
   if(++oks == n-1)
   then state_i := in_CS

when <done with CS>
   forall k∈pending_i send(k,ack);
   pending_i := ∅; state_i := dontcare;
```

```
when receive(k, req, req_num_k)
     C_i := max{C_i , req_num_k}+ 1;
     if(state_i==dontcare  or
        state_i== wait and
        (ticket_i ,i) > (ticket_k,k))
    then send(k,ack)
    else <add k in pending_i>
```

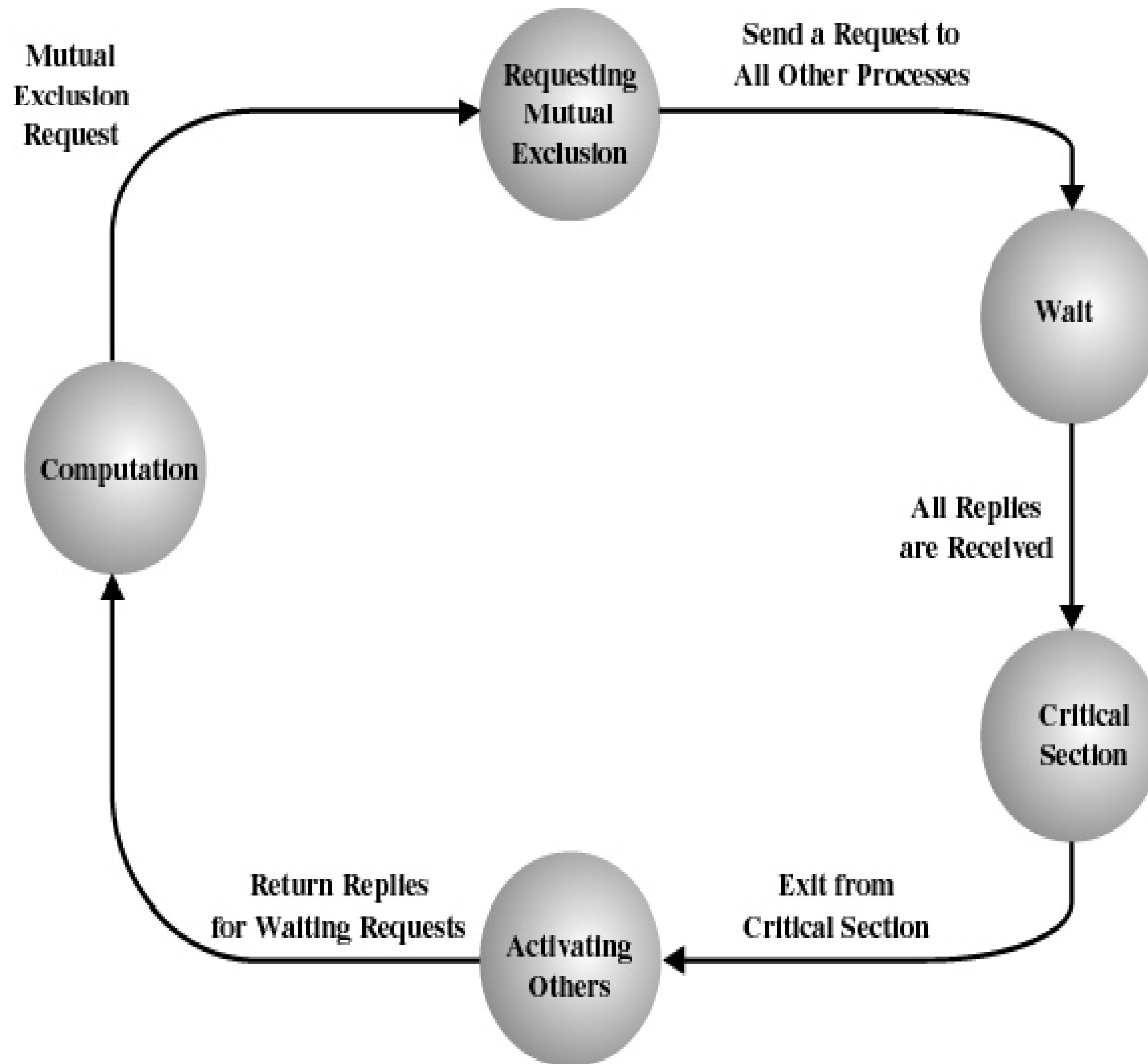


Figure 14.10 State Diagram for Algorithm in [RICA81]

Properties of last algo

- Mutex is guaranteed (prove by way of contradiction)
- Freedom from deadlock and starvation is ensured, since entry to the critical section is scheduled according to the ticket ordering, which ensures that
  - there always exists a process (the one with minimum ticket) which is able to enter its CS and
  - processes are served in a first-come-first-served order.

- The number of messages per critical-section entry is $2 \times (n - 1)$.

  (This is the minimum number of required messages per critical-section entry when processes act independently and concurrently.)

Method used: Event Ordering by Timestamping

- *Happened-before* relation (denoted by $\rightarrow$) on a set of events:
  - If A and B are events in the same process, and A was executed before B, then $A \rightarrow B$.
  - If A is the event of sending a message by one process and B is the event of receiving that message by another process, then $A \rightarrow B$.
  - If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$.

describing →: logical timestamps

- Associate a *timestamp* with each system *event*. Require that for every pair of events *A* and *B*:

  if $A \rightarrow B$, then the timestamp(*A*) < timestamp(*B*).

- Within *each* process P_i a *logical clock*, LC_i is associated: a simple counter that is:
  - incremented between any two successive events executed within a process.
  - advanced when the process receives a message whose timestamp is greater than the current value of its logical clock.

- If the timestamps of two events *A* and *B* are the same, then the events are concurrent. We may use the process identity numbers to break ties and to create a total ordering.

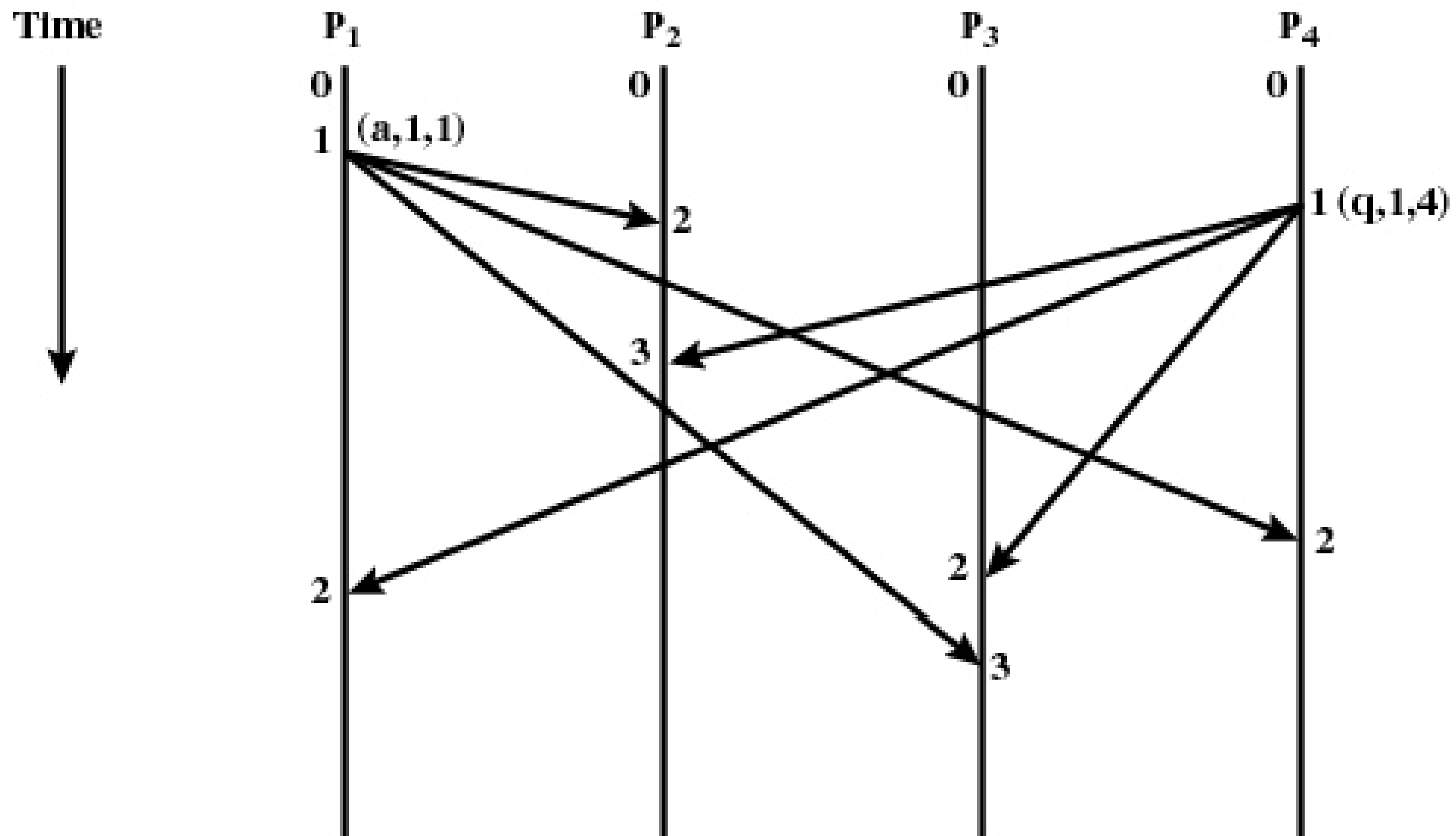


Figure 14.9 Another Example of Operation of Timestamping Algorithm

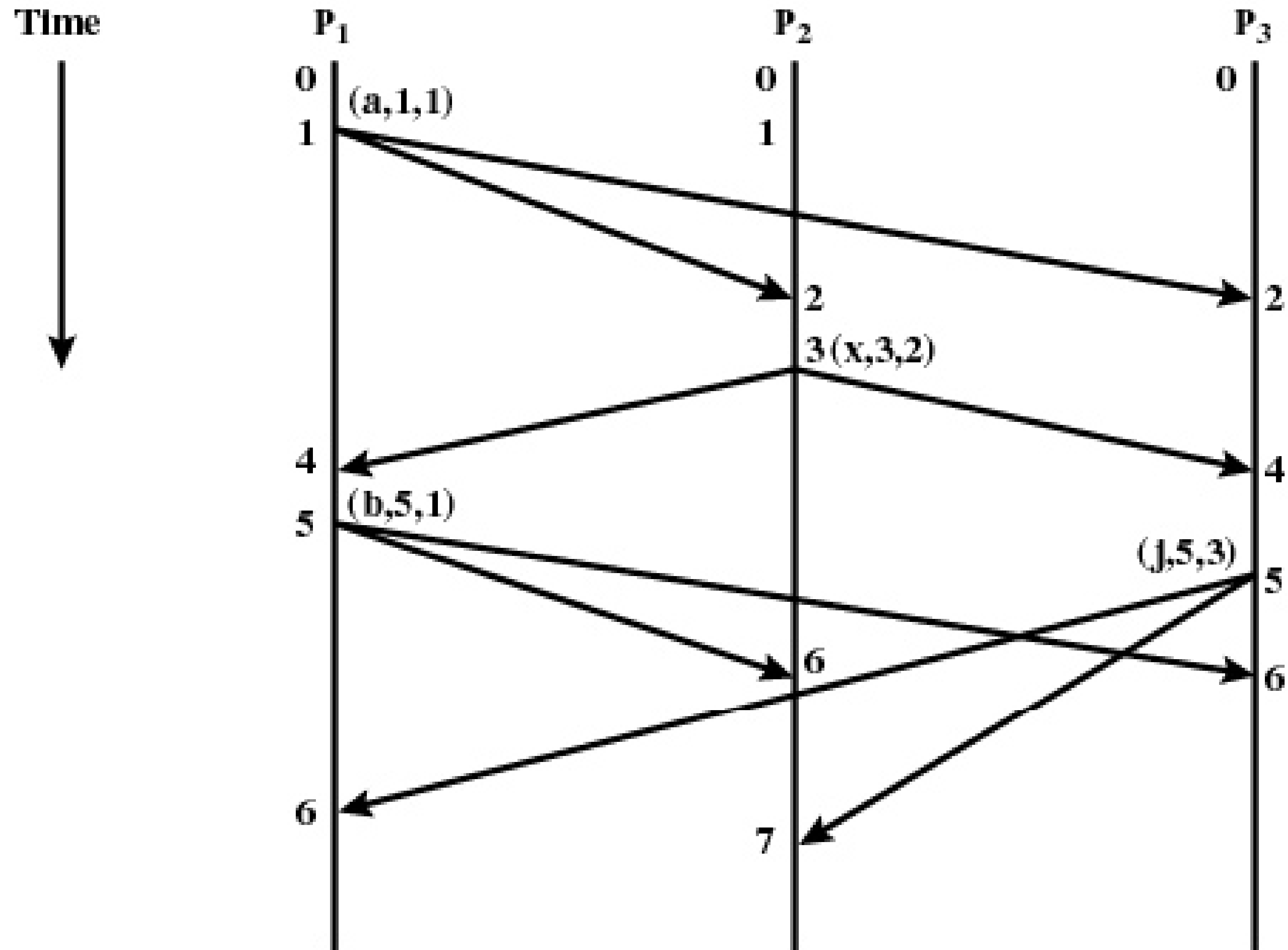


Figure 14.8 Example of Operation of Timestamping Algorithm