

Concurrency: Mutual Exclusion and Synchronization

Needs of Processes

- Allocation of processor time
- Allocation and sharing of resources (e.g. memory)
- Communication among processes
- Synchronization of multiple processes

Have seen: scheduling, memory allocation

Now: synchronization

Next: more on resource allocation & deadlock

Process Synchronization: Roadmap

- in *shared memory* systems
- in *message passing* systems



Shared memory communication

Recall:

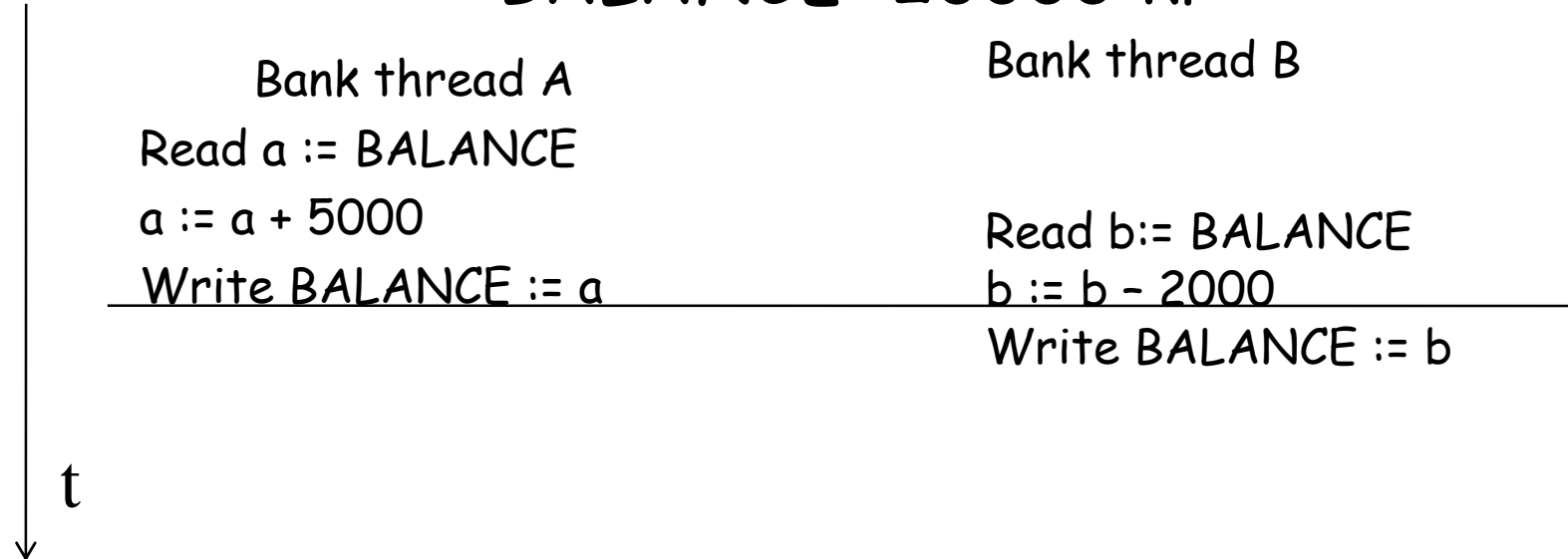
- Ex1: memory mapped files
- Ex2 (interface) POSIX Shared Memory:
 - define shared space, attach/detach

Btw, a nice link to check

www.cs.cf.ac.uk/Dave/C/node27.html#SECTION00272000000000000000

Money flies away ...

BALANCE: 20000 kr



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



- In **shared memory** systems
 - The **critical-section** (mutual exclusion - mutex) problem
 - Mutex for **2** and for **n** processes
 - Help from synchronization **hardware primitives**
 - Semaphores, Other common synchronization structures
 - Common synchronization problems
 - n process mutex revisited
 - Common OS cases (Linux, solaris, windows)
- 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

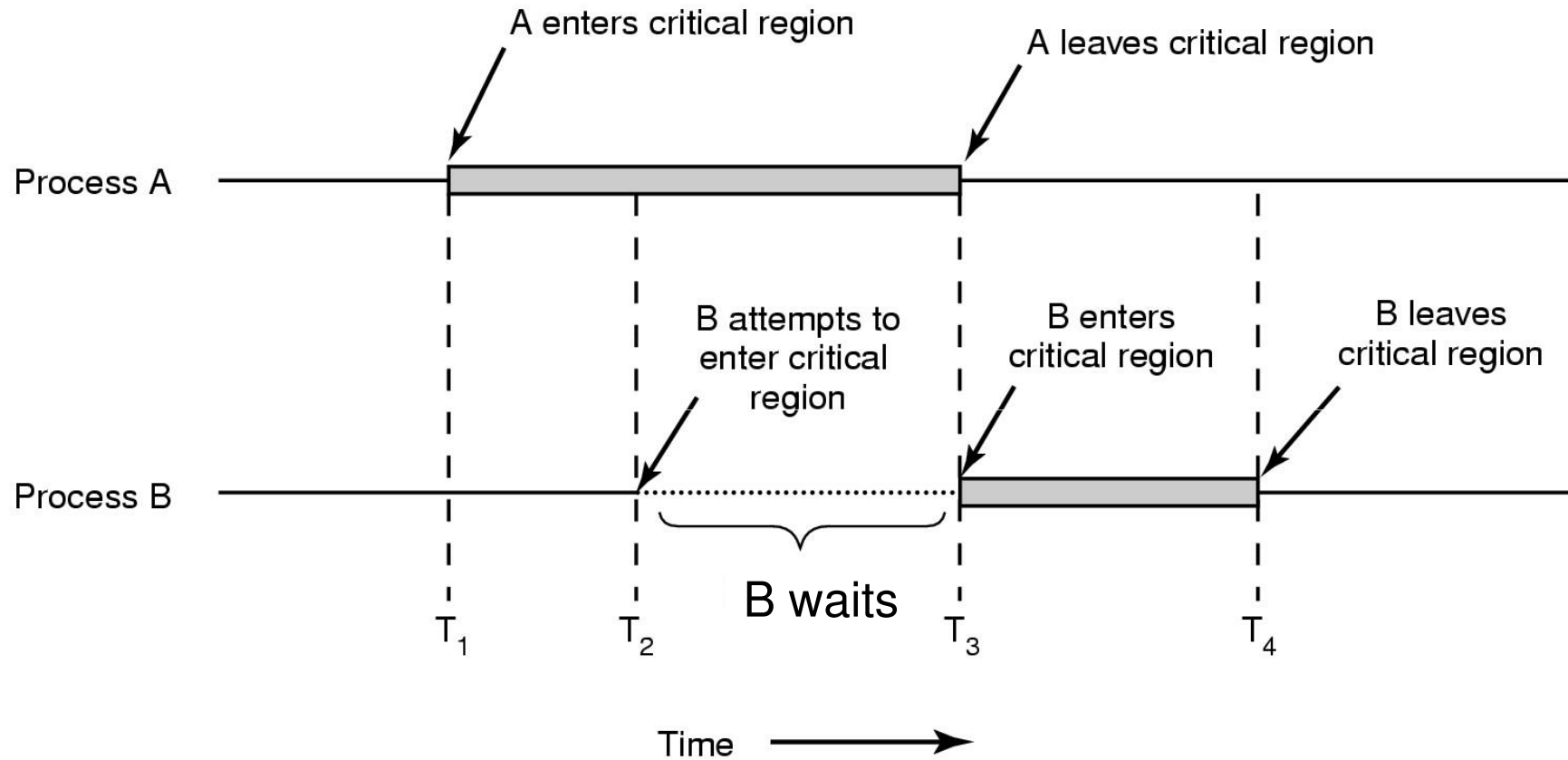
until false;

Requirements from a solution to the Critical-Section Problem



Bounded waiting,
Bounded # take-overs,
or similar

1. **Mutual Exclusion.** Only one process at a time is allowed to execute in its critical section.
2. **Progress (no deadlock/no livelock).** 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. **Fairness, Bounded Waiting (no starvation).** E.g. a bound 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.
 - No assumption concerning relative speed of the n processes.



Mutual exclusion/critical section (region)
Figure source A.Tanenbaum MOS book

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- Processes may share some common variables (can be read or written atomically) 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 ($turn = i \Rightarrow P_i$ can enter its critical section)

var *flag*: array [0..1] of boolean; initially false ($flag[i] = true \Rightarrow P_i$ 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;



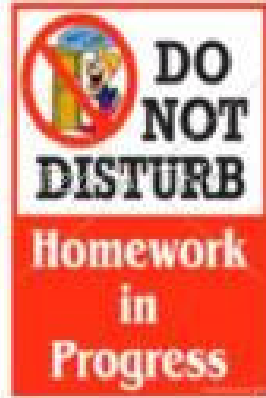
Argue that it satisfies the 3 requirements

Process Synchronization: Roadmap



- In **shared memory** systems
 - The **critical-section** (mutual exclusion - mutex) problem
 - Mutex for **2** (and for **n**) processes
 - - Help from synchronization **hardware primitives**
 - Semaphores, Other common synchronization structures
 - Common synchronization problems
 - n process mutex revisited
 - Common OS cases (Linux, solaris, windows)
- Synchronization in message passing systems

Mutual Exclusion: Hardware Support



A process runs until it invokes an operating-system service or until it is interrupted

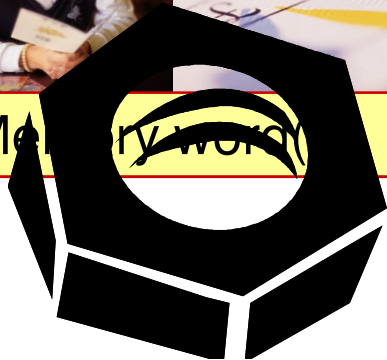
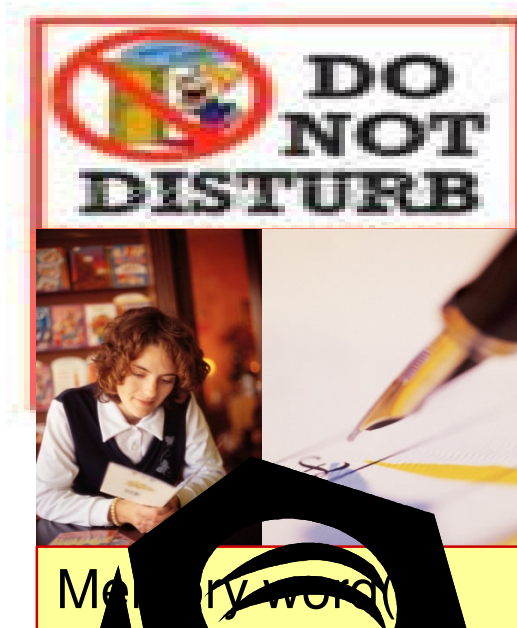
Interrupt Disabling disallows interleaving (1-cpu system) and can guarantee mutual exclusion

- 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 e.g. 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 (when used in the simple way shown just now)

- Busy-waiting consumes processor time
 - Even Deadlock possible if used in strict 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
- Starvation is possible when using just simple methods; cf next¹ for maintaining turn

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);



Process Synchronization: Roadmap



- In **shared memory** systems
 - The **critical-section** (mutual exclusion - mutex) problem
 - Mutex for **2** and for **n** processes
 - Help from synchronization **hardware primitives**
 - - Semaphores and Other common synchronization structures
 - Common synchronization problems
 - n process mutex revisited
 - Common OS cases (Linux, solaris, windows)
- Synchronization in message passing systems

Semaphores

- Special **variables/data-structures** 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: functionality

```
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;  
    }  
}
```

Binary and Counting semaphores: functionality

```
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 a queue;  
        block this process;  
    }  
}  
  
void signalB(binary_semaphore s)  
{  
    if (s.queue.is_empty())  
        s.value = 1;  
    else  
    {  
        remove a process from the 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 a queue;  
    }  
}  
  
void signal(semaphore s)  
{  
    s.count++;  
    if (s.queue.is_not_empty())  
    {  
        remove a process from the queue;  
        place process P on ready list;  
    }  
}
```

Notice: the queue and blocking behaviour not necessary; actually many semaphore implementations involve busy-waiting e.g. as in Peterson's algo or the TAS method

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
\vdots	\vdots
A	$wait(flag)$
$signal(flag)$	B

Watch for Deadlocks!!!

Let S and Q be two semaphores initialized to 1

P_0	P_1
$wait(S);$	$wait(Q);$
$wait(Q);$	$wait(S);$
\vdots	\vdots
$signal(S);$	$signal(Q);$
$signal(Q);$	$signal(S);$

Other synchronization constructs:

- Condition Variables
 - `condition x;`
 - Two operations possible on a condition variable:
 - `x.wait ()` - a process that invokes the operation is blocked.
 - `x.signal ()` - unblocks one of processes (if any) that invoked `x.wait ()` (if any; else, does nothing)
- Other high-level `synchronization constructs`
 - (conditional) critical regions (wait-until-value, ...)
 - monitors
 - Cf courses on parallelism

Process Synchronization: Roadmap



- In **shared memory** systems
 - The **critical-section** (mutual exclusion - mutex) problem
 - Mutex for **2** and for **n** processes
 - Help from synchronization **hardware primitives**
 - Semaphores and Other common synchronization structures
 - - Common synchronization problems
 - n process mutex revisited
 - Common OS cases (Linux, solaris, windows)
- Synchronization in message passing systems

Classical Problems of Synchronization

- Bounded-Buffer (producer-consumer)
- Dining-Philosophers (Resource allocation: we use as running example problem later, with deadlock avoidance)
- Readers and Writers (we use as running example later, with lock-free synch)

train on these: it is very useful and fun!

Bounded producer-consumer Buffer

- *Data*: N locations, each can hold one item
- Synchronization variables:
 - Binary semaphore **mutex** initialized to the value 1
 - General semaphore **avail-items** initialized to the value 0
 - General semaphore **avail-space** initialized to the value N .

producer process

```
do {  
    // produce an item  
  
    wait (avail-space);  
    wait (mutex);  
  
    // add the item to the buffer  
  
    signal (mutex);  
    signal (avail-items);  
  
} while (TRUE);
```

consumer process

```
do {  
    wait (avail-items)  
    wait (mutex);  
  
    // remove an item from buffer  
  
    signal (mutex);  
    signal (avail-space);  
  
    // consume the item  
  
} while (TRUE);
```

Process Synchronization: Roadmap



- In **shared memory** systems
 - The **critical-section (mutual exclusion - mutex)** problem
 - Mutex for **2** and for **n** processes
 - Help from synchronization **hardware primitives**
 - Semaphores and Other common synchronization structures
 - Common synchronization problems
 - - n process mutex revisited
 - Common OS cases (Linux, solaris, windows)
- Synchronization in message passing systems

Understanding synchronization better

Mutex for n processes using read/write variables

One idea (out several possible) : Before entering its critical section, each process receives a number. Holder of the smallest number enters the critical section.



Lamport's Bakery Algorithm

(Mutex for n processes using R/W variables)

Idea: Implement "nummerlappar" using read/write variables only

- numbering scheme may generate numbers in **non-decreasing order of enumeration**; i.e., 1,2,3,3,3,3,4,5
- 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.

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 $[i] := \max(\textit{number}[0], \textit{number}[1], \dots, \textit{number}[n-1]) + 1$;

choosing $[i] := false$;

for $j := 0$ to $n-1$ **do begin**

while *choosing* $[j]$ **do no-op**;

while *number* $[j] \neq 0$ and $(\textit{number}[j], j) < (\textit{number}[i], i)$ **do**
 no-op;

end;

 critical section

number $[i] := 0$;

 remainder section

until false;

*This is a more decentralized method:
uses no variable “writ-able” by all
processes*

Elaborate, think

- Argue why Bakery algorithm satisfies the 3 conditions for mutex
- Bakery algorithm idea not tied with R/W variables: how can it be implemented using e.g. semaphores?
- Having seen these, train on synchronization constructs, e.g:
 - implementing counting semaphores from binary ones,
 - semaphores using mutex solutions e.g. Peterson's, Lamport's, the Test-and-set method
 -

Process Synchronization: Roadmap



- In **shared memory** systems
 - The **critical-section (mutual exclusion - mutex)** problem
 - Mutex for **2** and for **n** processes
 - Help from synchronization **hardware primitives**
 - Semaphores and Other common synchronization structures
 - Common synchronization problems
 - n process mutex revisited
 - - Common OS cases (Linux, solaris, windows)
- Synchronization in message passing systems

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency (small critical sections)
 - adapt between busy waiting and blocking depending on contention
 - Blocked process queues are called turnstiles
- Uses **condition variables** and **readers-writers** locks when longer sections of code need access to data
 - **turnstiles** hold threads waiting on reader-writer lock and conditional variables as well.

Windows XP Synchronization

- Uses **interrupt masks** to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems and short critical sections
- Also provides **dispatcher objects** which may act as mutexes and semaphores
 - Dispatcher objects may provide **events** (similar to signal in a condition variable)

Linux Synchronization

- Linux provides:
 - semaphores
 - spin locks for multiprocessors and short critical sections
 - On uniprocessors: lock at kernel -level and kernel disables preemptions - again for short critical sections

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
 - mutex locks with blocking queues
 - condition variables
- Non-portable extensions include:
 - read-write locks
 - spin locks

Process Synchronization: Roadmap



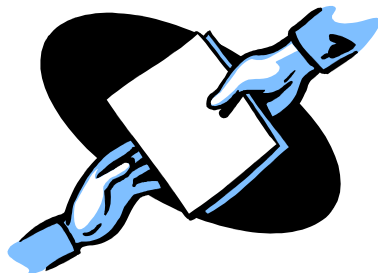
- In **shared memory** systems
 - The **critical-section** (mutual exclusion - mutex) problem
 - Mutex for **2** and for **n** processes
 - Help from synchronization **hardware primitives**
 - Semaphores and Other common synchronization structures
 - Common synchronization problems
 - n process mutex revisited
 - Common OS cases (Linux, solaris, windows)

→ Synchronization in message passing systems: mutex, coordination problems)

Synchronization using Message Passing communication

Recall:

- Mechanism for processes to **communicate and to synchronize (to some extent)** their actions, via
 - `send(message)`
 - `receive(message)`
- Can be direct or via mail box



Communication and synchronization with messages

Message passing may be

- **Blocking: synchronous**

- **Blocking send:** sender blocks until the message is received
- **Blocking receive:** receiver block until a message is available
- both blocking : **rendez-vous**



- **Non-blocking: asynchronous**

- **Non-blocking send:** the sender sends the message and continues
- **Non-blocking receive:** receiver receive a valid message or null
- can also have interrupt-driven receive



Mutual exclusion using messages: Centralized Approach



Key idea: One process 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));
}
```

Distributed Algorithms for mutex using messages

- 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 exists 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*;



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 $state_i \neq requesting$

$state_i := wait;$

$oks := 0;$

$req_num_i := ++C_i;$

forall k $send(k, req, req_num_i)$

when $receive(k, ack)$

if $(++oks == n-1)$

then $state_i := in_CS$

when $\langle done\ with\ CS \rangle$

forall $k \in pending_i$ $send(k, ack);$

$pending_i := \emptyset; state_i := dontcare;$



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 $\langle add\ k\ in\ pending_i \rangle$

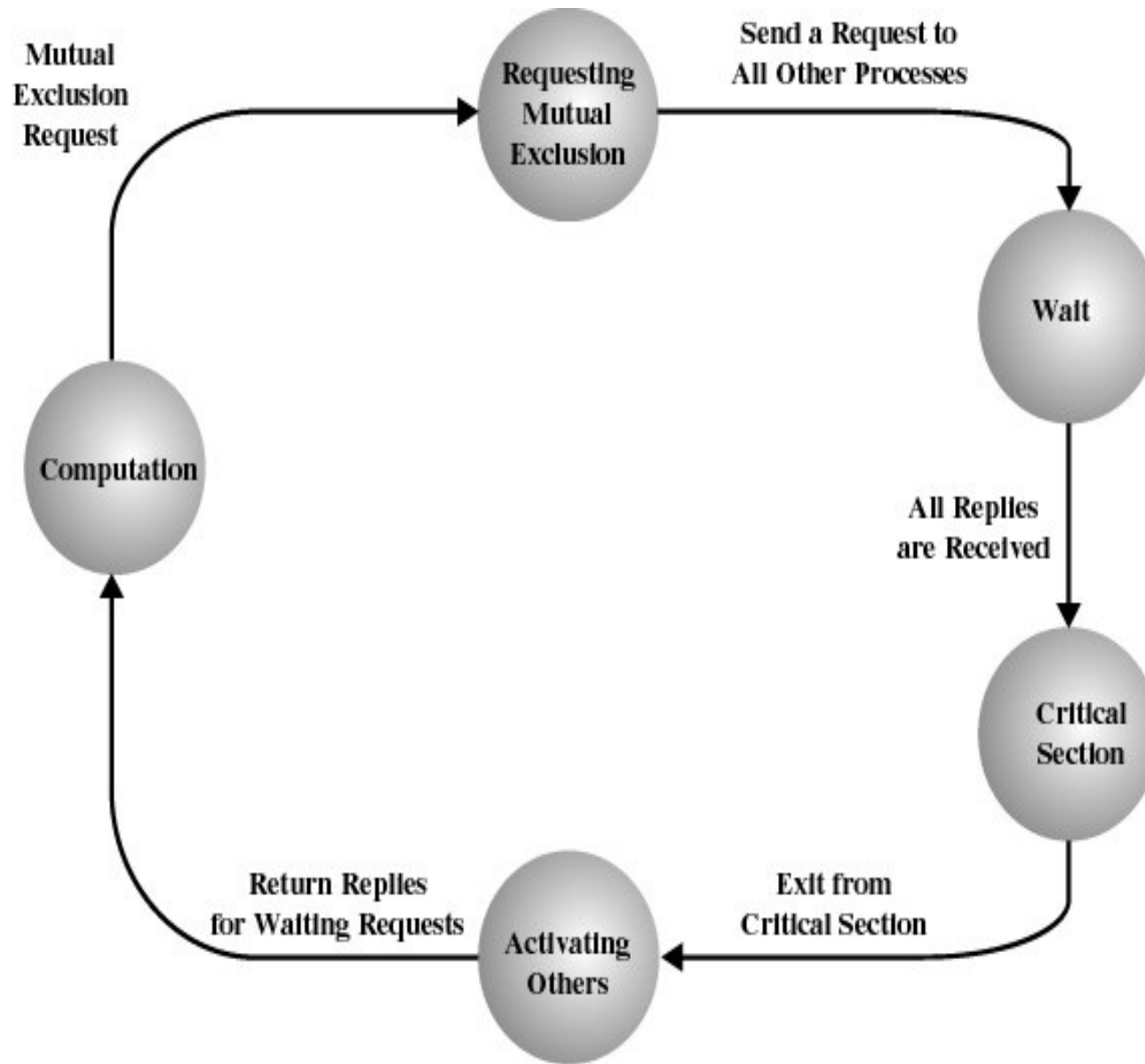


Figure 14.10 State Diagram for Algorithm in [RICA81]

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 \rightarrow : 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.

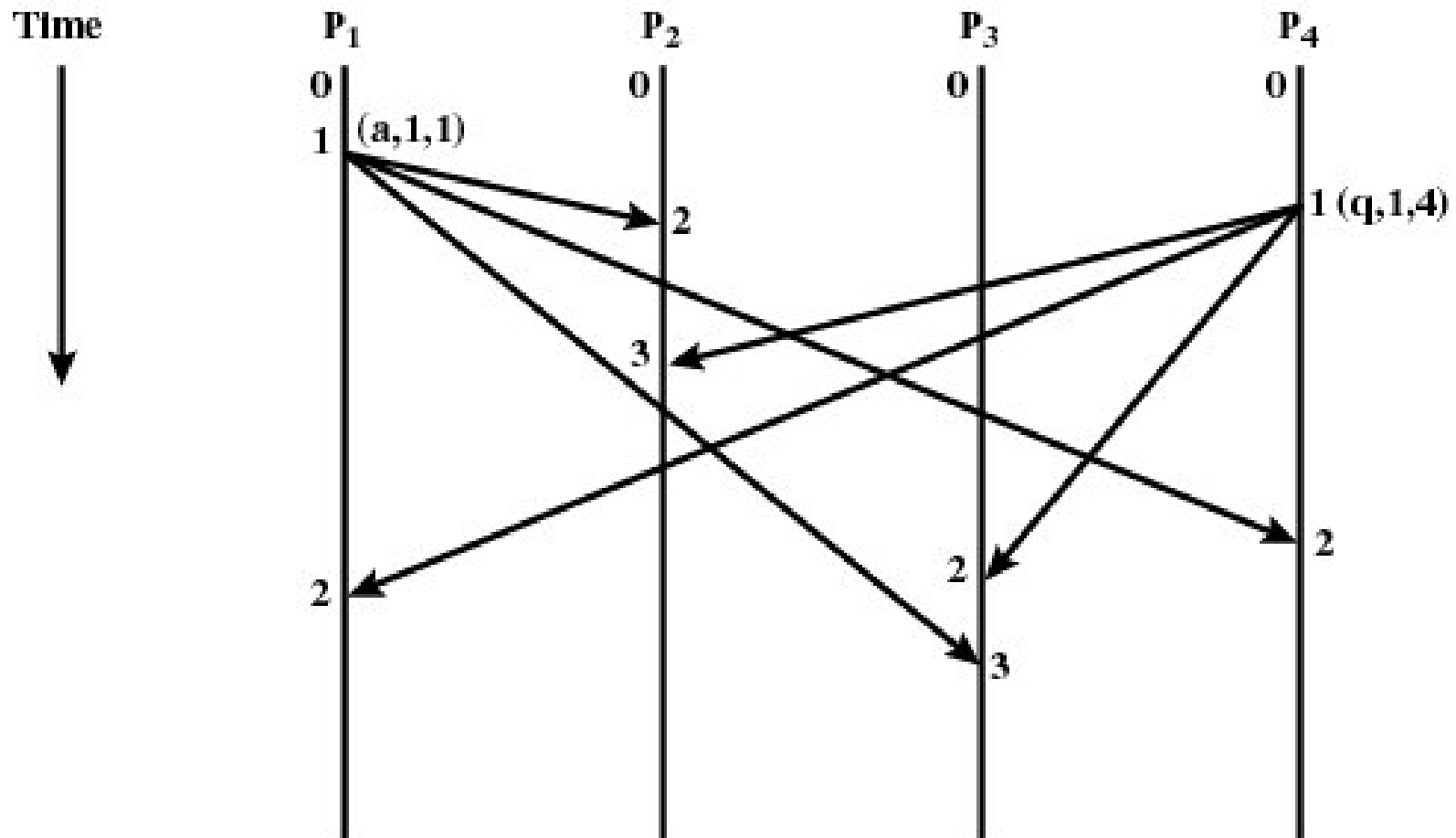


Figure 14.9 Another Example of Operation of Timestamping Algorithm

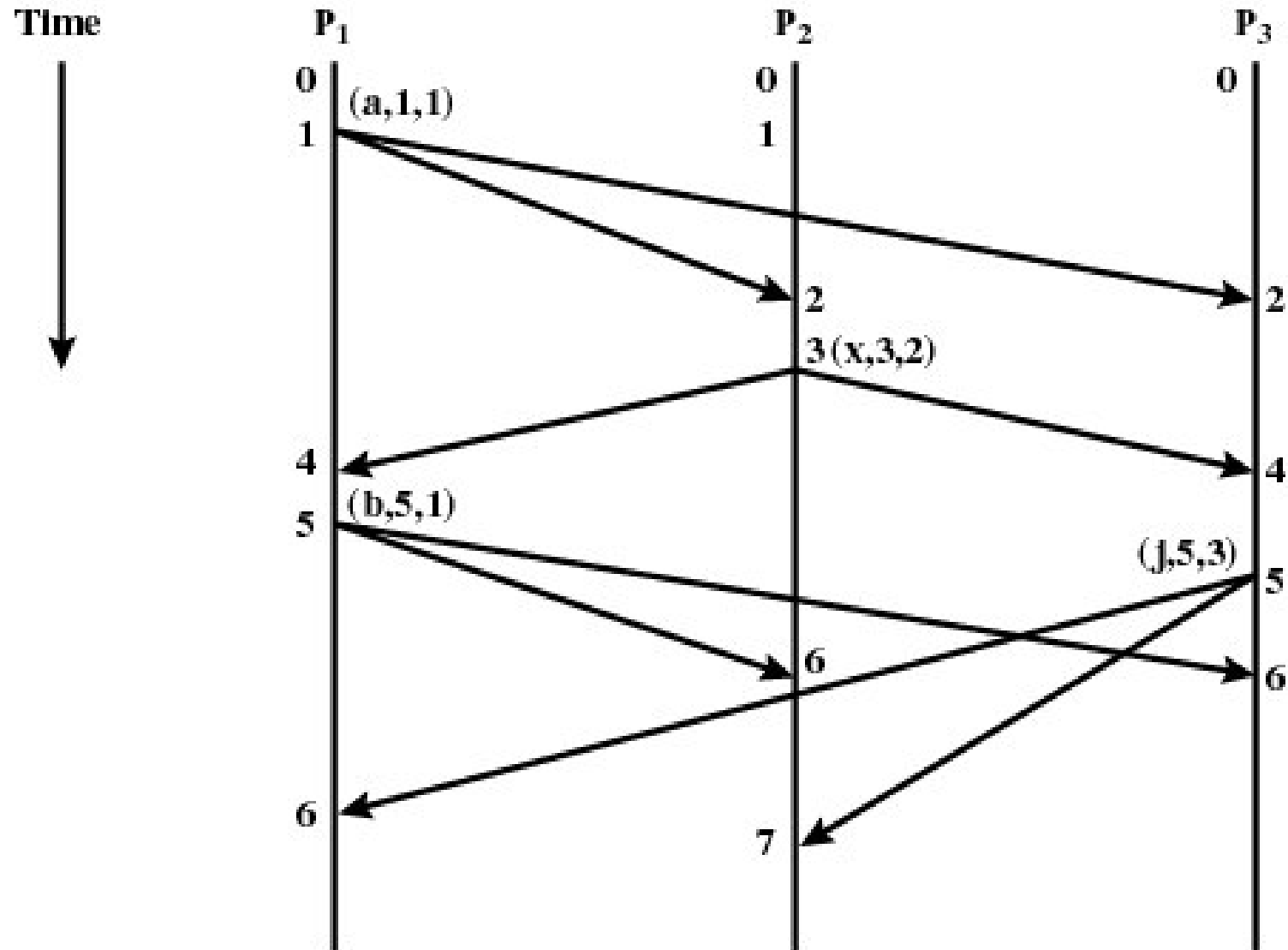


Figure 14.8 Example of Operation of Timestamping Algorithm

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

Key idea: similar as in the mailbox-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);
}
```