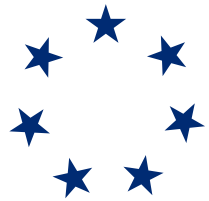
Synchronization

Instructor: Hengming Zou, Ph.D.

In Pursuit of Absolute Simplicity 求于至简，归于永恒

Content



- Why synchronization?
- Check and act
- Busy waiting

The gold fish problem

Low level synchronization

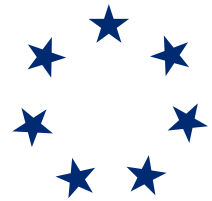
High level synchronization

- Lock
- Sleep/wake up
- Semaphore
- Monitor

The producer and consumer problem

- Message passing

Thread Synchronization

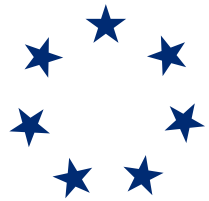


- Must control the inter-leavings between threads
  - Or the results can be non-deterministic

- Objective:
  - All possible inter-leavings must yield a correct answer
  - Try to constrain the thread executions as little as possible

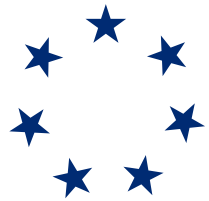
- Controlling the execution and order of threads is called
  - **SYNCHRONIZATION**

Gold Fish Problem



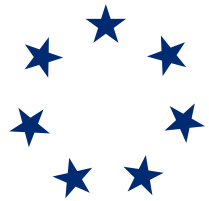
- Problem definition:
- Tracy and Peter want to keep a gold fish alive
  - By feeding the fish properly
- if either sees that the fish is not fed,
  - she/he goes to feed the fish
- The fish must be fed once and only once each day

Correctness Properties



- Someone will feed the fish if needed
  - No starvation
- But never more than one person feed the fish
  - No overeat

Solution #0



- No synchronization
  - Peter:
  - if (noFeed) {
  - feed fish
  - }

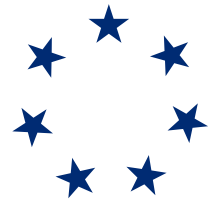
  Tracy:

  if (noFeed) {

  feed fish

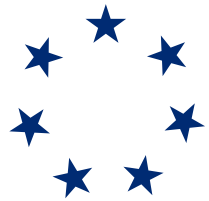
  }
- Peter — Tracy
- 3:00 look at the fish (no feed)
- 3:05 — look at the fish (no feed)
- 3:10 feed fish
- 3:25 — feed fish
- Fish over eat!

Problem with Solution #0



- Over eat occurred because:
  - They execute the same code at the same time
  - i.e. if (noFeed) feed fish
- This is called race
  - 2 or more threads try to access same shared resource at same time

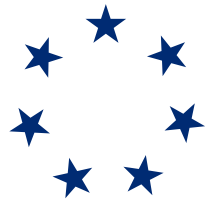
Critical Section



- The shared resource or code is called a critical section
  - the region where race condition occurs
- Access to it must be coordinated
  - i.e. only one thread can access it at a time

- In solution #0, critical section is
  - “if (noFeed) feed fish”
- Peter and Tracy must NOT be inside the CS at the same time

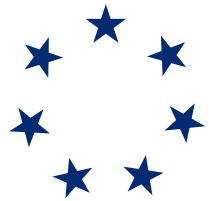
First Type of Synchronization



- Ensure only one person in critical section
  - i.e. only 1 person feeds the fish at a time
- This is called mutual exclusion:
  - only 1 thread is doing a thing at one time, others are excluded

- What conditions must mutual exclusion satisfy?
  - No two threads simultaneously in critical region
  - No thread running outside critical region may block another
  - No thread wait forever to enter critical region

Gold Fish Solution #1

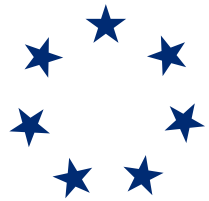


- Idea: leave note that you're going to check on the fish status,
  - so other person doesn't also feed

Peter:	Tracy:
if (noNote) {	if (noNote) {
leave note	leave note
if (noFeed) {	if (noFeed) {
feed fish	feed fish
}	}
remove note	remove note
}	}

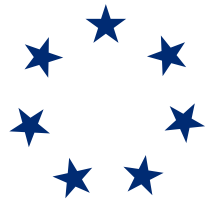
Does this work?

Solution #1



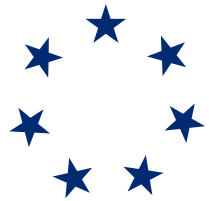
- No
- If not, when could it fail?
  - It does not prevent both parties entering CS at the same time
  - Thus failing the mutual exclusion requirement
- Is solution #1 better than solution #0?

Solution #2



- Idea:
  - change the order of “leave note” and “check note”
- This requires labeled notes
  - otherwise you’ll see your own note
  - and think it was the other person’s note

Solution #2



- Peter:
- leave notePeter
- if (no noteTracy) {
- if (noFeed) {
- feed fish
- }
- }
- remove notePeter

Tracy:

leave noteTracy

if (no notePeter) {

if (noFeed) {

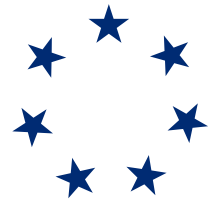
feed fish

}

}

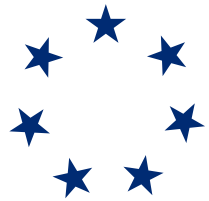
remove noteTracy

Solution #2



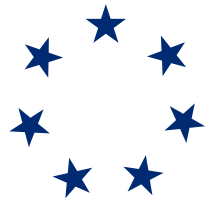
- Does this work?
  - Partially because it solves the over eat problem
- If not, when could it fail?
  - But it introduces starvation problem

Solution #3



- Idea:
  - have a way to decide who will feed fish
    - →when both leave notes at the same time.
- Approach:
  - Have Peter hang around to make sure job is done

Solution #3



- Peter:
- leave notePeter
- while (noTracy) {
- do nothing
- }
- if (noFeed) {
- feed fish
- }
- remove notePeter

Tracy:

leave noteTracy

if (no notePeter) {

if (noFeed) {

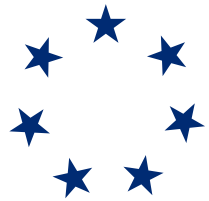
feed fish

}

}

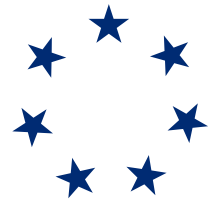
remove noteTracy

Solution #3



- Peter's "while (noteTracy)" prevents him from
  - running his critical section at same time as Tracy's
- It also prevents Peter from
  - running off without make sure that someone feeds fish

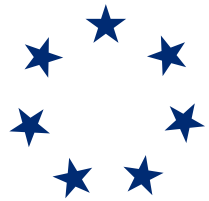
Proof of Correctness



- (Tracy) if no notePeter, then it's safe to feed
  - because Peter hasn't started yet.
- Peter will wait for Tracy to be done before checking fish status

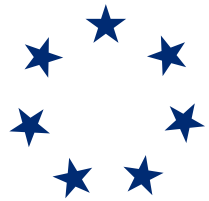
- (Tracy) if notePeter, then Peter is in the body of the code
  - and will eventually feed the fish (if needed)
- Note: Peter may be waiting for Tracy to quit

Proof of Correctness



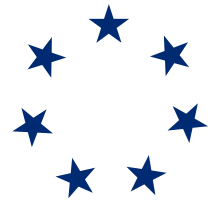
- (Peter) if no noteTracy, it's safe to feed
  - because Peter has already left notePeter, and
    - →Tracy will check notePeter in the future
- (Peter) if noteTracy, Peter hangs around and waits to see
  - if Tracy feeds fish
  - If Tracy feeds, we're done
  - If Tracy doesn't feed, Peter will feed

Solution #3



- Correct, but ugly
  - Complicated (non-intuitive) to prove correct
- Asymmetric
  - peter and Tracy runs different code
  - This makes coding difficult

Solution #3

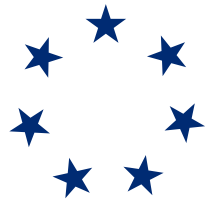


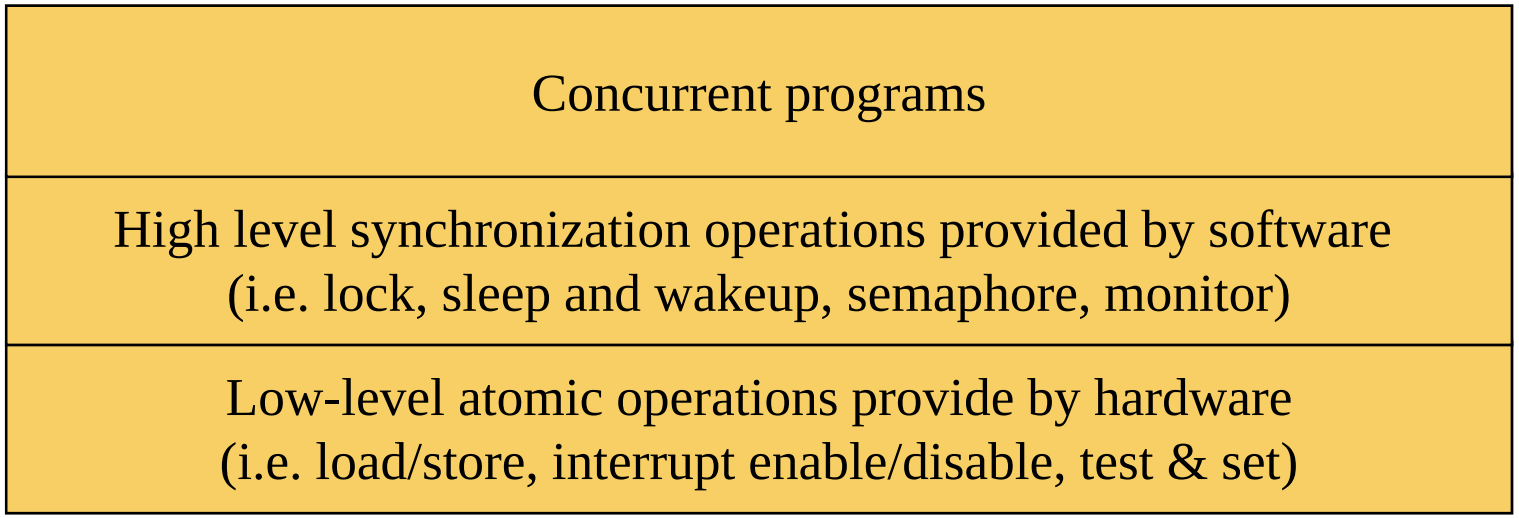
- Wasteful
  - Peter consumes CPU time while waiting for Tracy to remove note
- Constantly checking some status while waiting on something is called
  - busy-waiting
    - → Very bad

- What is the solution?
  - Raise the level of abstraction
- Make life easier for programmers

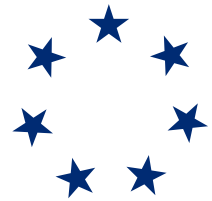
Higher-Level Synchronization

Higher-Level Synchronization





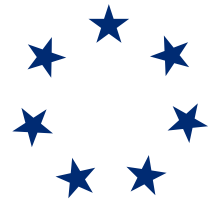
Lock (Mutex)



- A Lock prevents another thread from entering a critical section
  - e.g. lock the fish tank while you're checking
    - →so that both Peter and Tracy don't go feeding

- Two operations:
- lock(): wait until lock is free, then acquire it
  - do {
  - if (lock is free) {acquire lock; break}
  - } while (1)
- unlock(): release lock

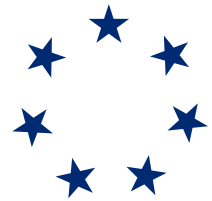
Lock (Mutex)



- What conditions must a lock satisfy?
  - Lock is initialized to be free
  - Acquire lock before entering critical section
  - Release lock when exiting critical section
  - Wait to acquire if another thread already holds it

- Why was the “note” in Gold Fish solutions #1 and #2 not a good lock?
- Does it meet the 4 conditions?

Solve “Gold Fish” with Lock



- Peter:

- lock()
- if (noFeed) {
- feed fish
- }
- unlock()

Tracy:

lock()
if (noFeed) {
feed fish
}
unlock()

- But this prevents Tracy from doing stuff while Peter is feeding.
  - I.e. critical section includes the feeding time.
- How to minimize the time the lock is held?

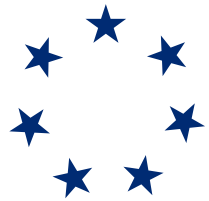
Producer-Consumer Problem

- Problem:
  - producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization for coordinating producer and consumer

Producer-Consumer Problem

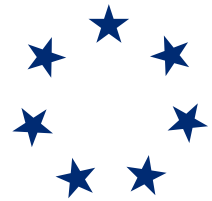
- Buffer between producer and consumer allows them to
  - operate somewhat independently
- Otherwise must operate in lockstep
  - producer puts 1 thing in buffer, then consumer takes it out
  - then producer adds another, then consumer takes it out, etc

Producer-Consumer Problem



- Coke machine example:
- Delivery person (producer) fills machine with cokes
- Students (consumer) feed cokes and drink them
- Coke machine has finite space (buffer)

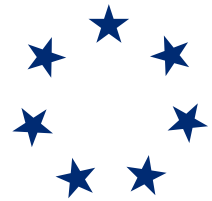
Solution for PC Problem



- What does producer do when buffer full?
- What does consumer do when buffer empty?
- The busy waiting solution in Solution 3 is unacceptable

- Use Sleep and Wakeup primitives
- Consumer sleeps if buffer is empty
  - Wake up sleeping producers otherwise
- Producer sleeps if buffer is full
  - Wake up sleeping consumers otherwise

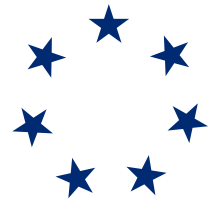
Sleep and Wakeup



```
#define N 100      //# of slots in the buffer
int count=0;       //# of items in the buffer

void producer(void)
{     int item;
      while(TRUE) {
            item=produce_item();
            if(count==N) sleep();
            insert_item(item);
            count=count+1;
            if(count==1) wakeup(consumer);
      }
}
```

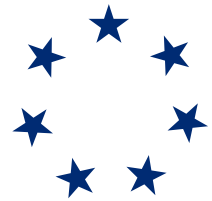
```
void consumer(void)
{     int item;
      while(TRUE) {
            if(count==0) sleep();
            item=remove_item();
            count=count-1;
            if(count==N-1) wakeup(producer);
            consume_item(item);
      }
}
```

What are the problem?



- Producer-consumer problem with fatal race condition
  - Access to “count” is un-restrained
  - Wakeup call could get lost
- Solution is to use semaphore

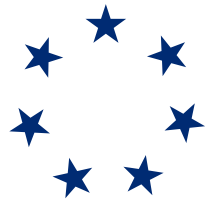
Semaphore



- Semaphores are like a generalized lock
- It has a non-negative integer value
- Semaphore supports the following ops:
  - down, up

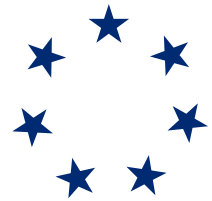
- **Semaphore is both**
  - **a synchronization primitive and**
  - **an IPC mechanism**

Semaphore



- Down Operation:
  - Wait for semaphore to become positive
  - Then decrement semaphore by 1
    - →Originally called “P” operation for the Dutch proberen
- Up Operation:
  - Increment semaphore by 1
    - →originally called “V”, for the Dutch verhogen
  - This wakes up a thread waiting in down
    - →if there are any

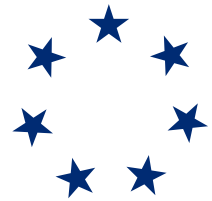
Semaphores



- Can also set the initial value of semaphore
- The key parts in down() and up() are atomic
- Two down() calls at the same time can’t decrement the value below 0

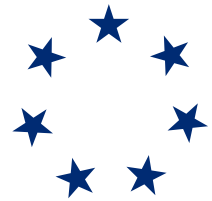
- Semaphore not only can remember the number of signals
- It can also be used as a lock
  - If it is allowed only value 0 and 1, which is called binary semaphore

Binary Semaphores



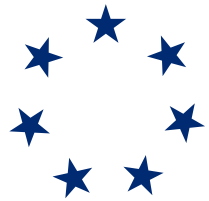
- Value is either 0 or 1
- down() waits for value to become 1
  - then sets it to 0
- up() sets value to 1
  - waking up waiting down (if any)

Mutual Exclusion w/ Semaphore



- Initial value is 1 (or more generally, N)
  - down()
  - <critical section>
  - up()
- Like lock/unlock, but more general

Semaphores for Ordering



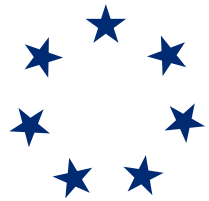
- Usually (not always) initial value is 0
- Example: thread A wants to wait for thread B to finish before continuing
  - Semaphore initialized to 0
  - A
  - down()
  - continue execution

  B

  do task

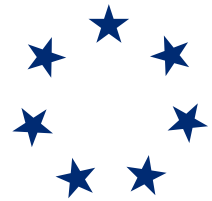
  up()

PC Problem with Semaphores



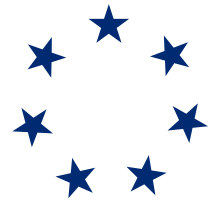
- Let’s solve the producer-consumer problem in semaphores
- Semaphore Assignments:
  - mutex:
    - →ensures exclusion around code that manipulates buffer queue
    - →(initialized to 1)
  - full:
    - →counts the # of full buffers (initialized to 0)
  - empty:
    - →counts the number of empty buffers (initialized to N)
- Why do we need different semaphores for full and empty buffers?

Solve PC with Semaphores



```
#define N 100                    //# of slots in the buffer
typedef int semaphore;           //a special kind of int
semaphore mutex =1;              //controls access to critical
semaphore empty =N;              //counts empty buffer slots
int full = 0;                    //counts full buffer slots
```

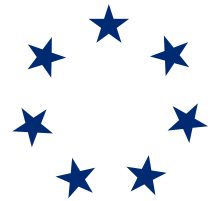
Solve PC with Semaphores



```
void producer(void)
{
    int item;
    while(TRUE) {
        item=produce_item();
        down(empty);
        down(mutex);
        insert_item(item);
        up(mutex)
        up(full);
    }
}
```

```
void consumer(void)
{
    int item;
    while(TRUE) {
        down(full);
        down(mutex);
        item=remove_item();
        up(mutex);
        up(empty);
        consume_item(item);
    }
}
```

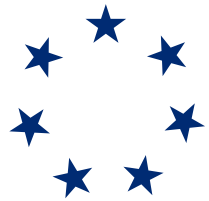
Semaphore Assignments



- Does the order of the down() function calls matter in the consumer
  - (or the producer)?
- Does the order of the up() function calls matter in the consumer
  - (or the producer)?

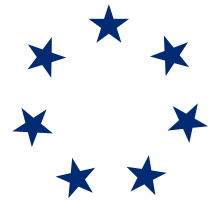
- What (if anything) must change to:
  - allow multiple producers and/or multiple consumers?
- What if there's 1 full buffer, and
  - multiple consumers call down(full) at the same time?

Problem with Semaphore



- Is there any problem with semaphore?
- The order of down and up ops are critical
  - Improper ordering could cause deadlock
  - i.e. programming in semaphore is tricky
- How to make programming easier?

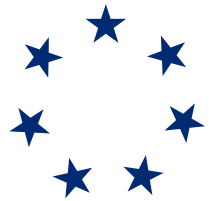
Monitor



- A programming language construct
- A collection of procedures, variables, data structures
- Access to it is guaranteed to be exclusive
  - By the compiler (not programmer)

- Monitors use separate mechanisms for the two types of synchronization
  - Use locks for mutual exclusion
  - Use condition variables for ordering constraints
- **monitor = a lock + the condition variables associated with that lock**

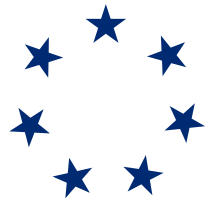
Condition Variables



- Main idea:
  - make it possible for thread to sleep inside a critical section
- Approach:
  - by atomically releasing lock, putting thread on wait queue and sleep

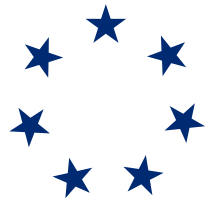
- Each variable has a queue of waiting threads
  - threads that are sleeping, waiting for a condition
- Each variable is associated with one lock

Monitors



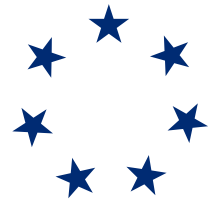
```
begin monitor
    integer i;
    condition c;
    procedure producer();
    ...
    end;

    procedure consumer();
    ...
    end
end monitor
```

Producer-Consumer in Monitor



```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert (item: integer);
        begin
            if count == N then wait(full);
            insert_item(item); count++;
            if count ==1 then signal(empty);
        end;
    function remove: integer;
        begin
            if count ==0 then wait(empty);
            remove = remove_item; count--;
            if count==N-1 then signal(full);
        end;
count:=0;
end monitor;
```

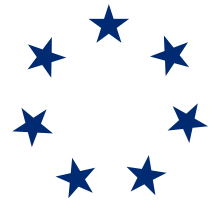
Producer-Consumer in Monitor



```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item);
    end
end;
```

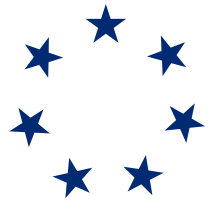
```
procedure consumer;
begin
    while true do
    begin
        item=ProducerConsumer.remove;
        consume_item(item);
    end
end;
```

Ops on Condition Variables



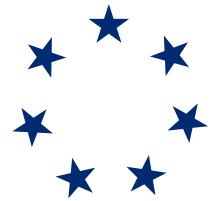
- wait():
  - atomically release lock
  - put thread on condition wait queue, go to sleep
  - i.e. start to wait for wakeup
- signal():
  - wake up a thread waiting on this condition variable if any
- broadcast():
  - wake up all threads waiting on this condition variable if any
- Note that thread must be holding lock when it calls wait() or signal()
- To avoid problems when both threads are inside monitor
  - the signal() must be the last statement

Ops on Condition Variables



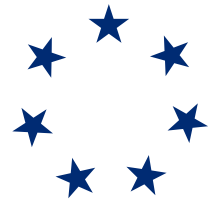
- What to do when a thread wakes up?
- Two options:
- Let the wakeup thread run
  - i.e. signaler release the lock
- Let the caller and callee compete for lock

Mesa vs. Hoare Monitors



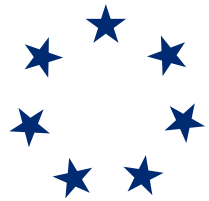
- The first option is called Hoare Monitor
  - It gives special priority to the woken-up waiter
  - signaling thread gives up lock
  - woken-up waiter acquires lock
  - signaling thread re-acquires lock after waiter unlocks
- The second option is Mesa Monitor
  - when waiter is woken, it must contend for the lock with other threads
  - hence must re-check condition
  - Whoever wins get to run
- **We'll stick with Mesa monitors because it is more flexible**
  - **As most operating systems do**

Compare Monitors and Semaphores



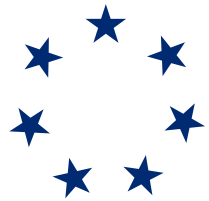
- Do not show this slide
- Semaphores used for both mutual exclusion and ordering constraints
  - elegant (one mechanism for both purposes)
  - code can be hard to read and hard to get right
- Monitor lock is just like a binary semaphore that is initialized to 1
  - lock() = down()
  - unlock() = up()

Condition Variable vs. Semaphore



Condition Variable	Semaphores
While (cond) {wailt()};	Down()
Conditional code in user program	Conditional code in semaphore definition
User writes customized condition	Condition specified by semaphore definition (wait if value ==0)
User provides shared variables, protect with lock	Provides shared variable & thread-safe operations on that integer
No memory of past signals	“Remembers” past up() calls

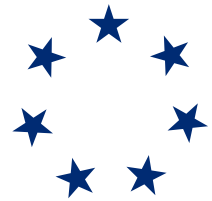
Condition Variable vs. Semaphore



- Condition variables are more flexible than
  - using semaphores for ordering constraints
- Condition variables:
  - can use arbitrary condition to wait
- Semaphores:
  - wait if semaphore value == 0

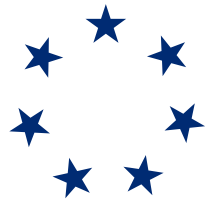
- Semaphores work best if
  - the shared integer and waiting condition (==0) maps
    - →naturally to the problem domain

Problems with Monitor



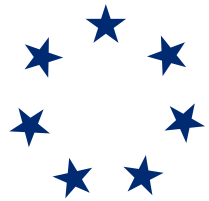
- Many languages do not support
- Don’t resolve problem when multiple CPUs or computer are involved
  - This applies to semaphore too
- Solution?
  - Message passing

Message Passing



- A high-level primitive for IPC
- It uses two primitives: send and receive
  - Send(destination, &message)
  - Receive(source, &message)
- They are system calls (not language constructs)
- Can either block or return immediately

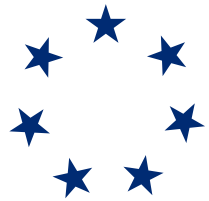
Message Passing



```
#define N 100      //# of slots in the buffer
void producer(void)
{     int item;
      message m;   //message buffer
      while(TRUE) {
            item=produce_item();
            receive(consumer, &m);
            build_messasge(&m, item);
            send(consumer, &m);
      }
}
```

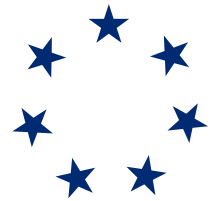
```
void consumer(void)
{     int item;
      message m;
      for(i=0;i<N;i++)
            send(producer, &m);  //send N slots
      while(TRUE) {
            receive(producer, &m);
            item=extract_item(&m);
            send(producer, &m);
            consume_item(item);
      }
}
```

Problems with Message Passing



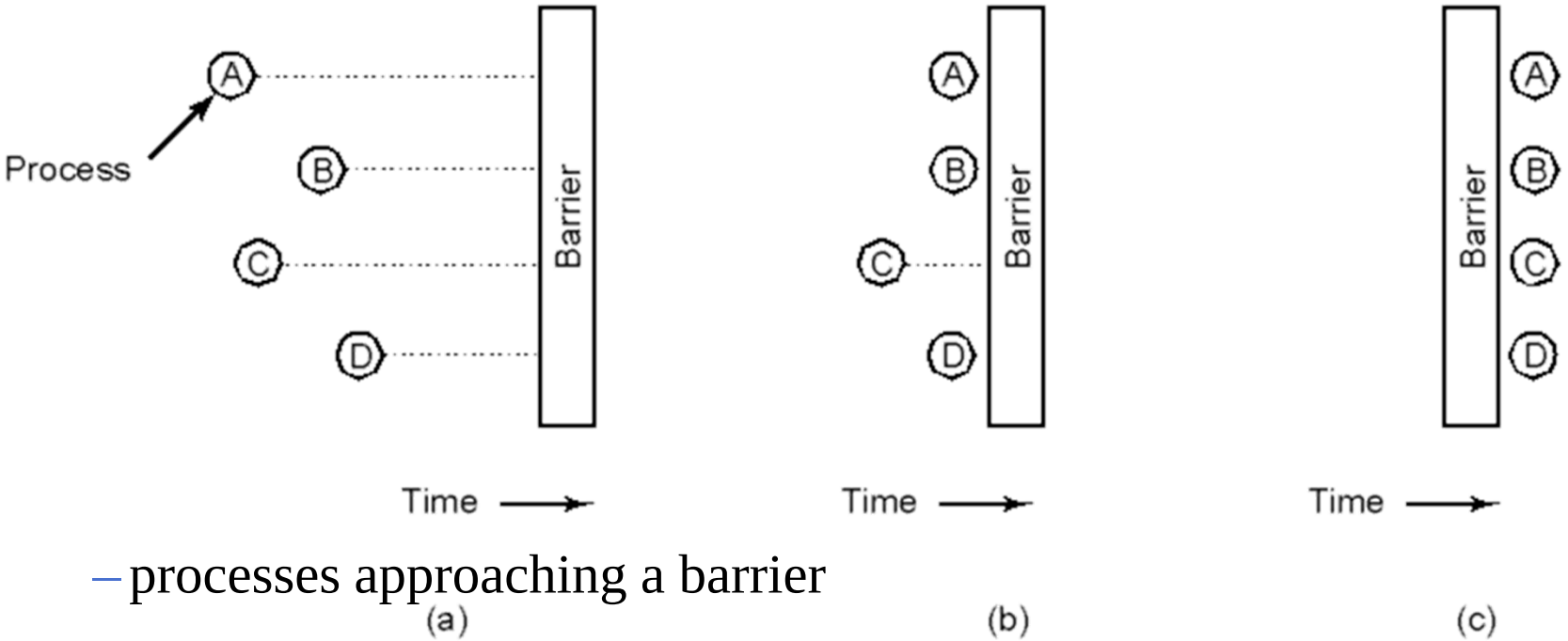
- Have many challenging problems
- Message loss
  - What happens when a message is lost
- Authentication
  - How to determine the identity of the sender?
- Performance

Barriers



- Another synchronization primitive
- Intended for a group of processes
- All processes must reach the barrier
  - for the applications to proceed to next phase

Barriers



- processes approaching a barrier
- all processes but one blocked at barrier
- last process arrives, all are let through


Thoughts Change Reality
意念改变现实