Operating Systems

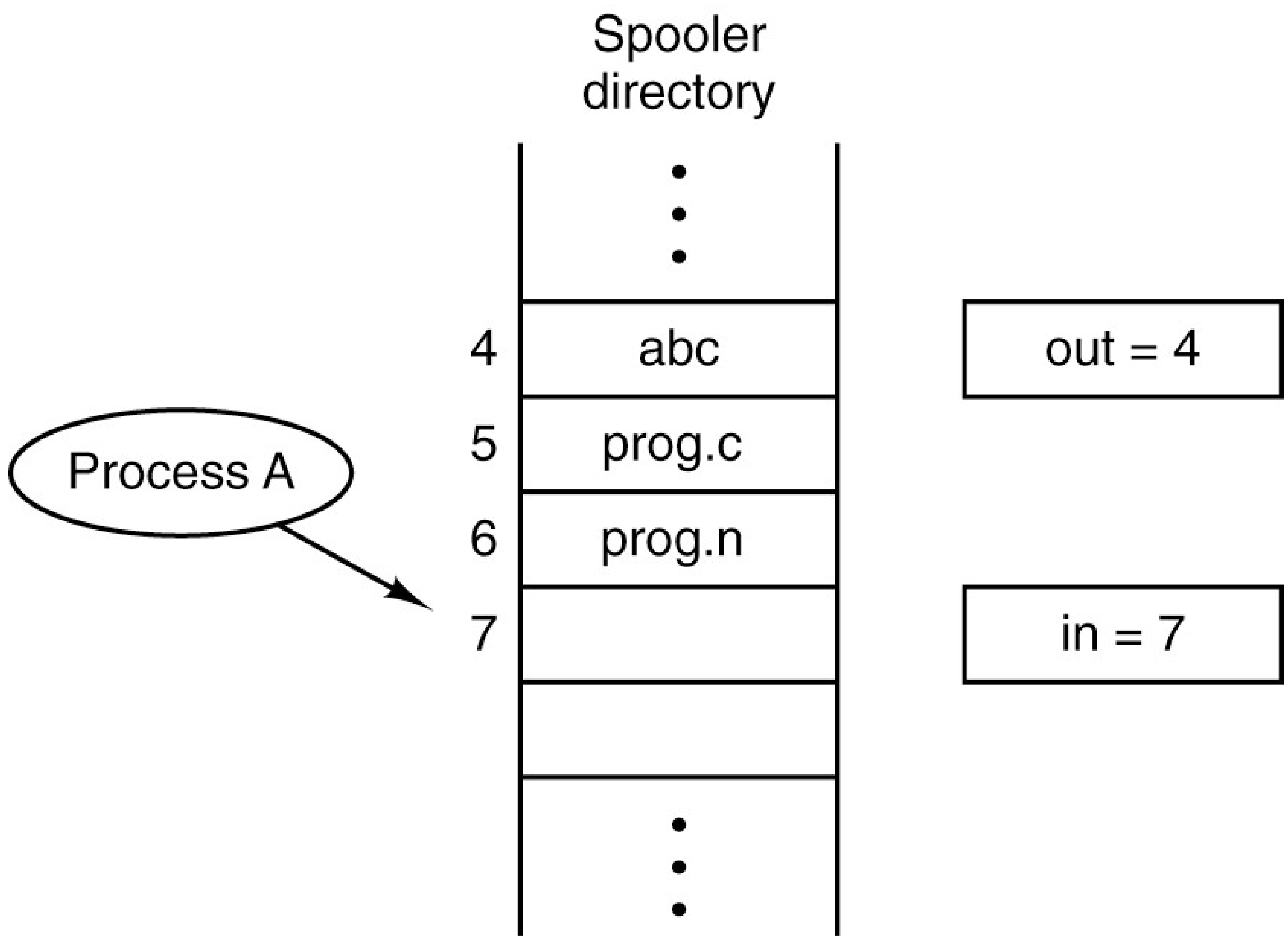
Race Conditions

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

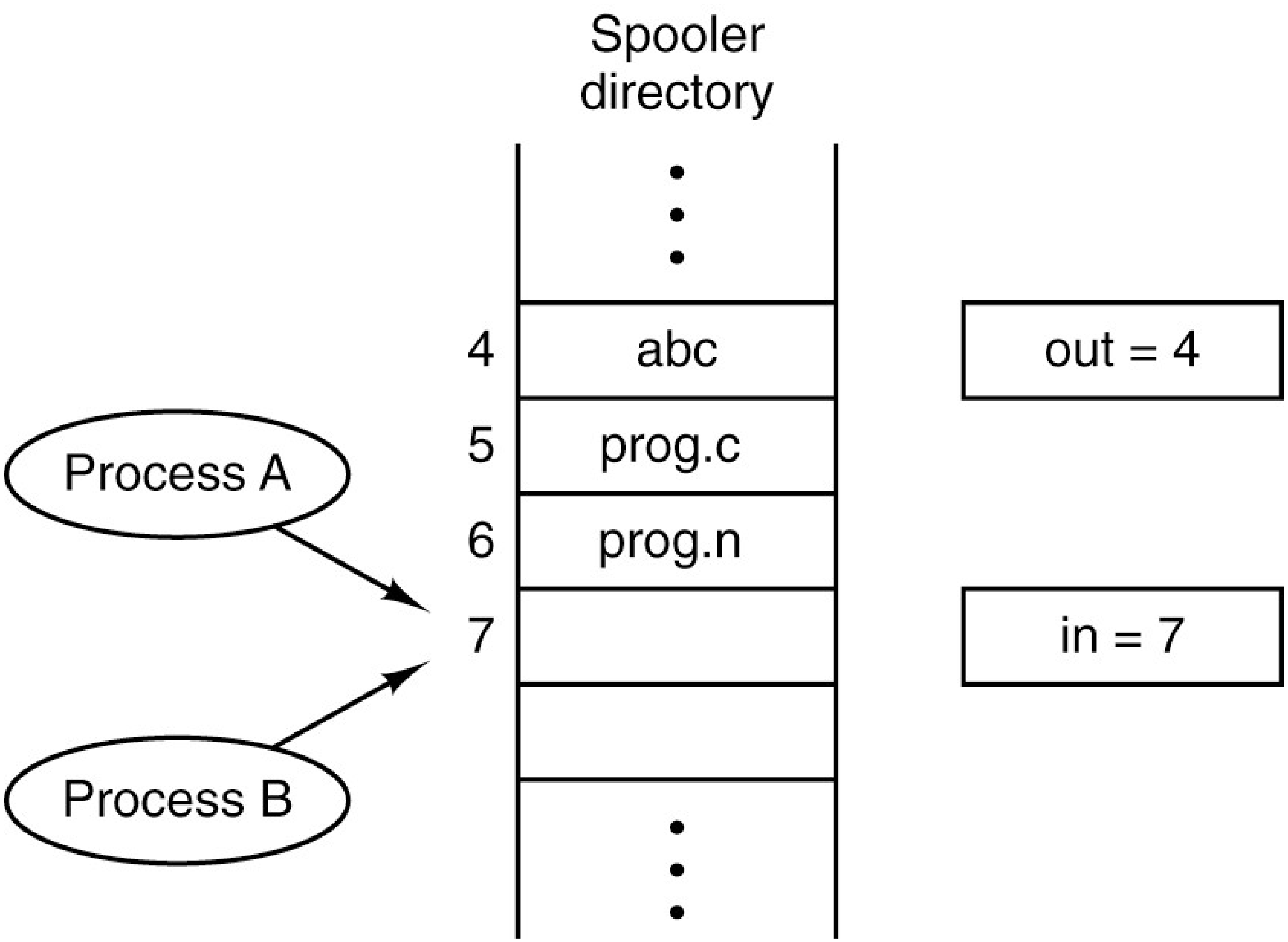
Interprocess Communication (IPC)

- Processes frequently need to communicate with other processes
- Three main issues:
  - How can one process pass information to another?
  - Need to make sure two or more processes do not get in each other's way.
  - Ensure proper sequencing when dependencies exist

Example of IPC

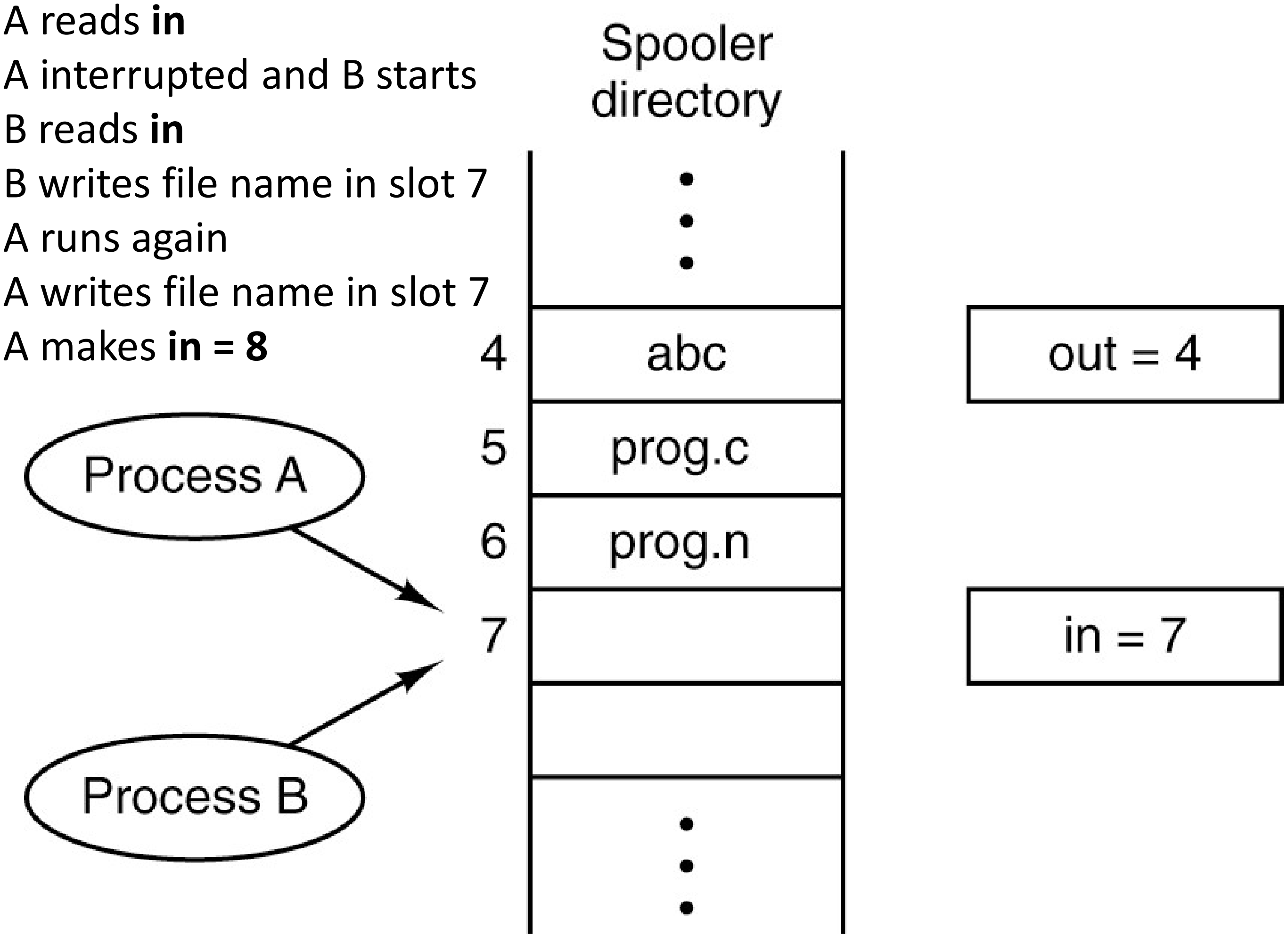


Example of IPC



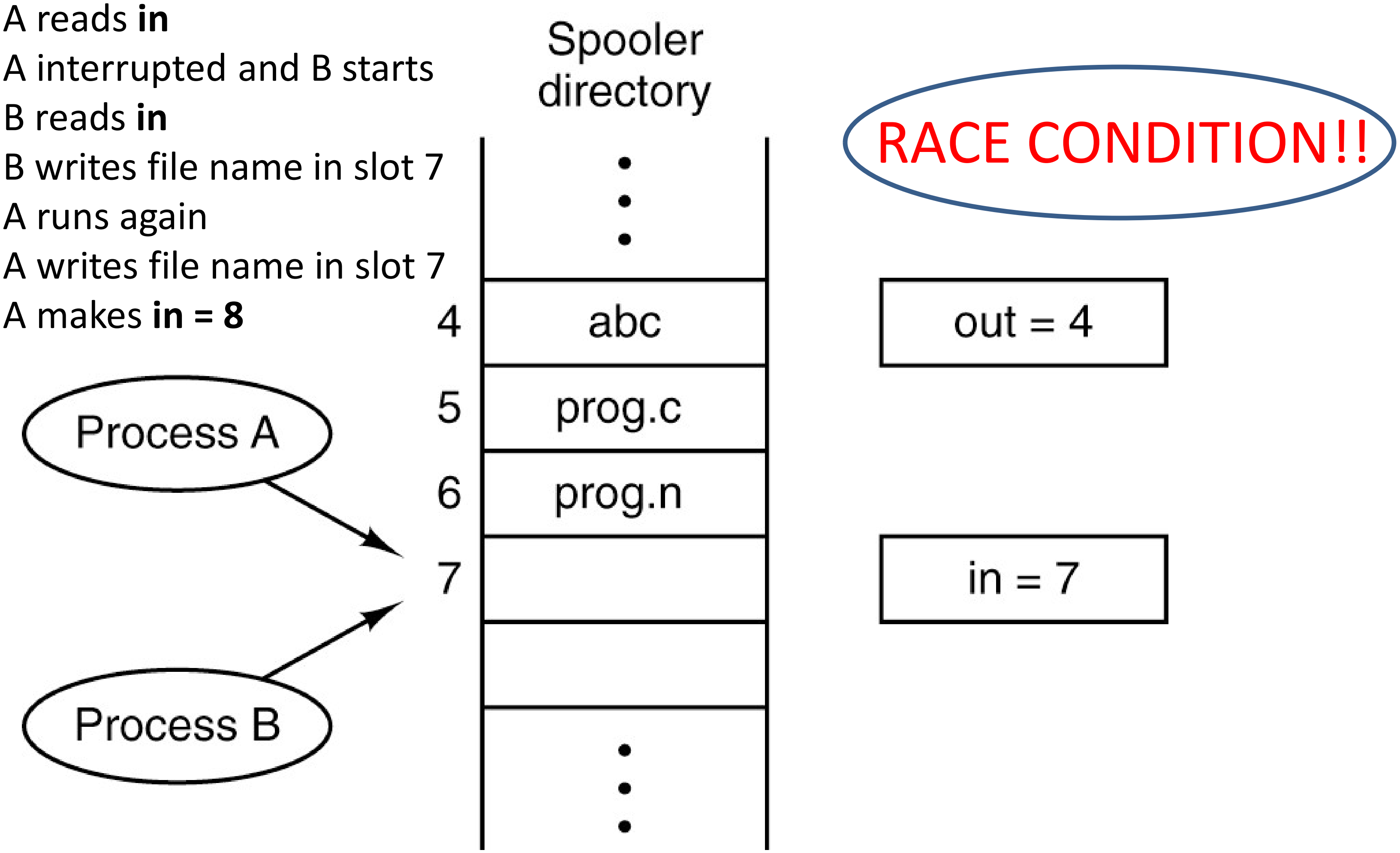
Example of IPC

1. Process A reads **in**
2. Process A interrupted and B starts
3. Process B reads **in**
4. Process B writes file name in slot 7
5. Process A runs again
6. Process A writes file name in slot 7
7. Process A makes **in = 8**



Example of IPC

1. Process A reads **in**
2. Process A interrupted and B starts
3. Process B reads **in**
4. Process B writes file name in slot 7
5. Process A runs again
6. Process A writes file name in slot 7
7. Process A makes **in = 8**

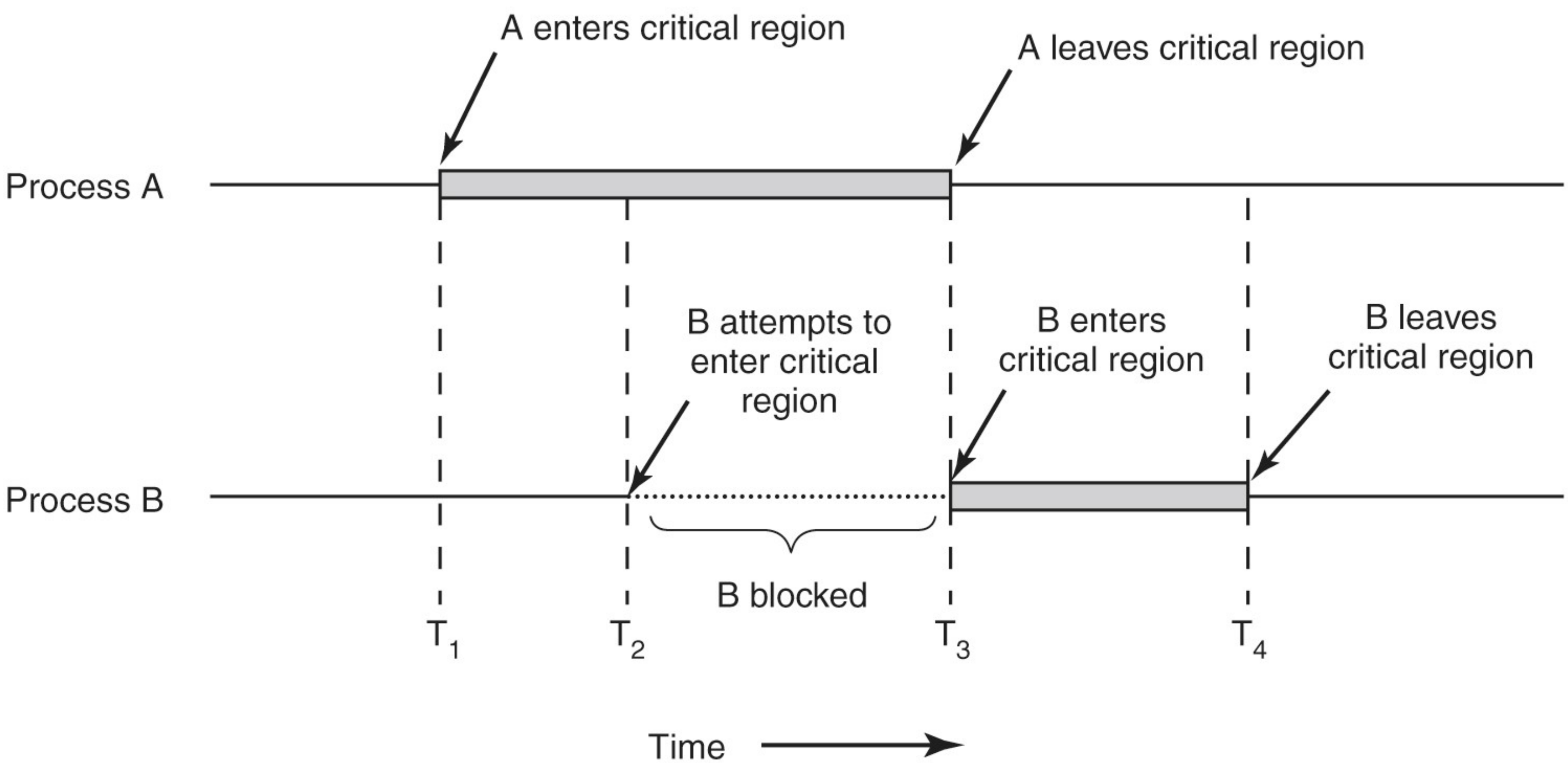


How to Avoid Race Condition?

- Prohibit more than one process from reading and writing the shared data at the same time -> mutual exclusion
- The choice of appropriate primitive operations for achieving mutual exclusion is a major design issue in an OS
- The part of the program where the shared memory is accessed is called the critical region

Conditions of Good Solutions

1. No two processes may be simultaneously inside their critical region.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process has to wait forever to enter its critical region.


A enters critical region
A leaves critical region
Process A
B attempts to enter critical region
B enters critical region
B leaves critical region
Process B
B blocked
T_1
T_2
T_3
T_4
Time

Solution 1:
Disabling Interrupts

Have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it.

Solution 1: Why is it Bad?

- Unwise to give user processes the power to turn off interrupts
- Affects only one CPU and not other CPUs in the system in case of multicore or multiprocessor systems

Solution 2:
Lock Variables

Have a shared (lock) variable, initially set to 0. When a process wants to enter its critical region, it first tests the lock:

- If 0, the process sets it to 1 and enters the critical region
- If 1, process waits until it becomes 0

Solution 2:
Why is it Bad?

- Process A reads the lock and finds it 0
- Before it can set it to 1, process A is stopped and process B starts
- Process B finds the lock to be 0, so it sets it to 1 and enters the critical region
- Process B is stopped and process A runs
- Process A sets the lock to 1 and enters the critical region

Solution 2: Why is it Bad?

- Process A reads the lock and finds it 0
- Before it can set it to 1, process A is stopped and process B starts
- Process B finds the lock to be 0, so it sets it to 1 and enters the critical region
- Process B is stopped and process A runs
- Process A sets the lock to 1 and enters the critical region

Two processes will be in the critical region at the same time!!

Solution 3: Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

Process 0

Variable turn is initially 0

Process 1

Solution 3: Strict Alternation

Busy waiting

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

Variable turn is initially 0

Process 0

Process 1

Solution 3:
Strict Alternation: Why Bad?

What if process 0 is much faster than process 1?

```
while (TRUE) {
      while (turn != 0)        /* loop */ ;
      critical_region( );
      turn = 1;
      noncritical_region( );
}
```

Process 0

```
while (TRUE) {
      while (turn != 1)        /* loop */ ;
      critical_region( );
      turn = 0;
      noncritical_region( );
}
```

Process 1

- Process 1 spends a lot of time here!
- Process 0 finishes its part and sets turn to 1
- Process 1 is stuck in noncritical region and prohibits process 0 from entering the critical region.

Solution 3:
Strict Alternation: Why Bad?

What if process 0 is much faster than process 1?

```
while (TRUE) {
      while (turn != 0)          /* loop */ ;
      critical_region( );
      turn = 1;
      noncritical_region( );
}
```

```
while (TRUE) {
      while (turn != 1)          /* loop */ ;
      critical_region( );
      turn = 0;
      noncritical_region( );
}
```

Process 0

Process 1

Violating condition 3!!

Taking turn is not a good idea when one of the processes is much slower than the other.

Solution 4:
Peterson's Solution

process 0	process 1
enter_region(0)	enter_region(1)
Critical Section	Critical Section
leave_region(0);	leave_region(1);

Solution 4: Peterson's Solution

```
#define FALSE  0
#define TRUE   1
#define N      2                            /* number of processes */

int turn;                                   /* whose turn is it? */
int interested[N];                          /* all values initially 0 (FALSE) */

void enter_region(int process);             /* process is 0 or 1 */
{
    int other;                              /* number of the other process */

    other = 1 – process;                    /* the opposite of process */
    interested[process] = TRUE;             /* show that you are interested */
    turn = process;                         /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)              /* process: who is leaving */
{
    interested[process] = FALSE;            /* indicate departure from critical region */
}
```

Hardware Solution

- The instruction: TSL RX, LOCK
  - TSL = Test and Set Lock
  - Reads the content of memory word *lock* into register RX, and then stores a nonzero value into *lock*
  - The whole operation is atomic

Hardware Solution

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was nonzero, lock was set, so loop
        RET                        | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET                        | return to caller
```

Similar Hardware Solution

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
        CMP REGISTER,#0           | was lock zero?
        JNE enter_region          | if it was non zero, lock was set, so loop
        RET                       | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0              | store a 0 in lock
        RET                       | return to caller
```

About Previous Solutions

- Processes must call enter_region and leave_region in the correct timing. If a process cheats, the mutual exclusion will fail.
- The main drawbacks of all these solutions is busy waiting. Keeping the CPU busy doing nothing is not the best thing to do.
  - Wastes CPU time
  - Priority inversion problem (process of higher priority has to wait for a process of lower priority).

Sleep and Wakeup

- IPC primitives
- Block instead of wasting CPU time
- Two system calls:
  - sleep: causes the caller to block until another process wakes it up
  - wakeup: has one parameter, the process to be awakened

First Let's see the: Producer Consumer Problem

- Two processes share a common fixed size buffer
- One process (producer): puts info into the buffer
- The other process (consumer): removes info from the buffer

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

void producer(void)
{
     int item;

     while (TRUE) {                                     /* repeat forever */
          item = produce_item( );                       /* generate next item */
          if (count == N) sleep( );                     /* if buffer is full, go to sleep */
          insert_item(item);                            /* put item in buffer */
          count = count + 1;                            /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);             /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                                     /* repeat forever */
          if (count == 0) sleep( );                     /* if buffer is empty, got to sleep */
          item = remove_item( );                        /* take item out of buffer */
          count = count – 1;                            /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer);         /* was buffer full? */
          consume_item(item);                           /* print item */
     }
}
```

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

void producer(void)
{
     int item;

     while (TRUE) {                              /* repeat forever */
          item = produce_item( );                /* generate next item */
          if (count == N) sleep( );              /* if buffer is full, go to sleep */
          insert_item(item);                     /* put item in buffer */
          count = count + 1;                     /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);      /* was buffer empty? */
     }
}
```

What happens if consumer() stopped after reading count (=0) ?

LOST WAKEUP PROBLEM

```
void consumer(void)
{
     int item;

     while (TRUE) {                              /* repeat forever */
          if (count == 0) sleep( );              /* if buffer is empty, got to sleep */
          item = remove_item( );                 /* take item out of buffer */
          count = count - 1;                     /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer);  /* was buffer full? */
          consume_item(item);                    /* print item */
     }
}
```

How to Solve The Lost Wakeup Problem?

- Add a wakeup waiting bit to the picture
  - When a wakeup is sent to a process that is still awake, this bit is set.
  - Later, when the process tries to go to sleep and the bit is set, the bit will be reset but the process will remain awake.
- BUT: What happens when we have more than two processes? How many bits shall we use?

Better Solution for Lost Wakeup Problem: **Semaphores**

- Integer to count the number of wakeups saved for future use
- Two primitives: down and up
    - atomic actions

down: if value = 0 then sleeps

otherwise, decrements it and continue

up: increments the value, and wakes up a sleeping process (if any)

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

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

Mutexes??

- A variable that can be in one of two states: locked and unlocked
- Can be used to manage critical sections
- Managed using TSL or XCHG

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Didn't We Say Processes Do Not Share Address Space?

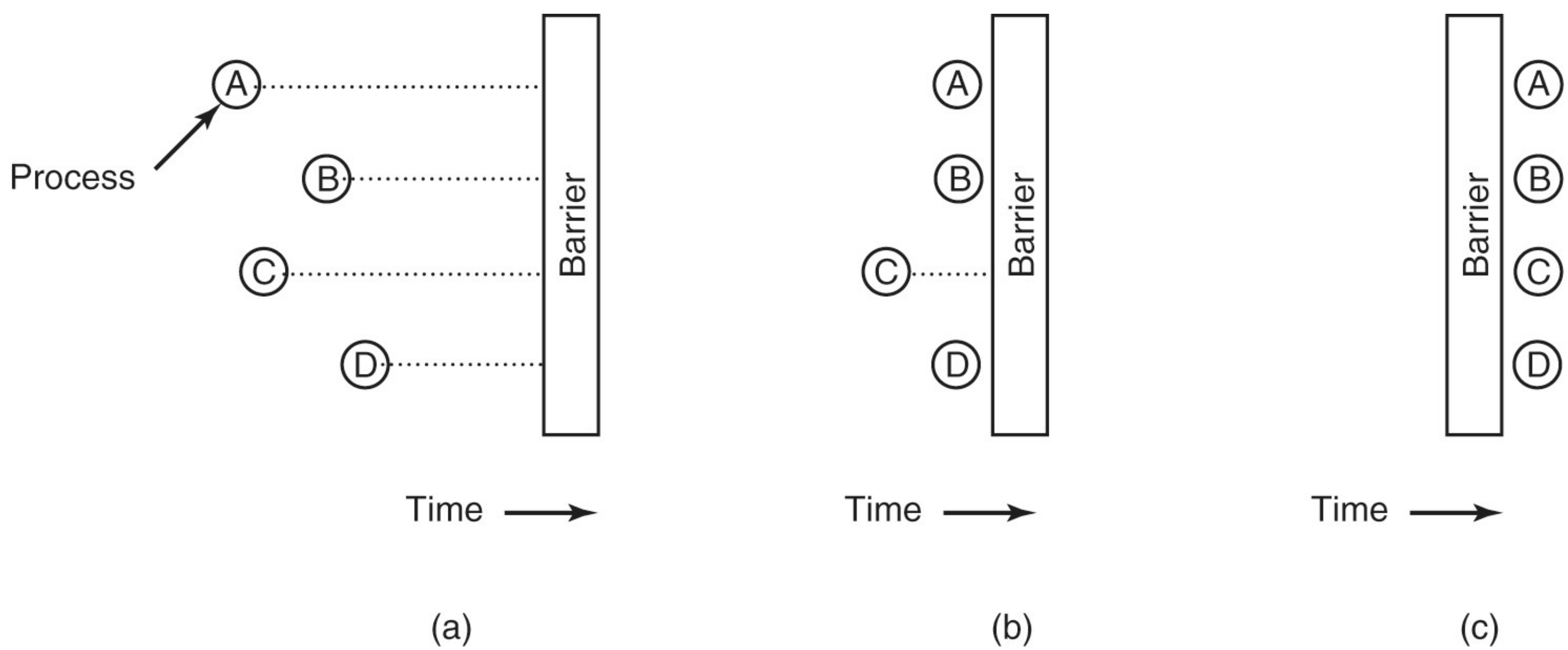
- Some of the shared data structures can be stored in the kernel and accessed through system calls.
- Most modern OSes offer ways for processes to share some portions of their address spaces with other processes

Forget About Sharing: How About Message Passing?

- Two primitives: send and receive
- May be used across machines
- Are system calls
  - send(destination, &message)
  - receive(source, &message)
- Issues
  - Lost acknowledgement
  - Authentication
  - performance (message passing is always slower than stuff like semaphores, …)

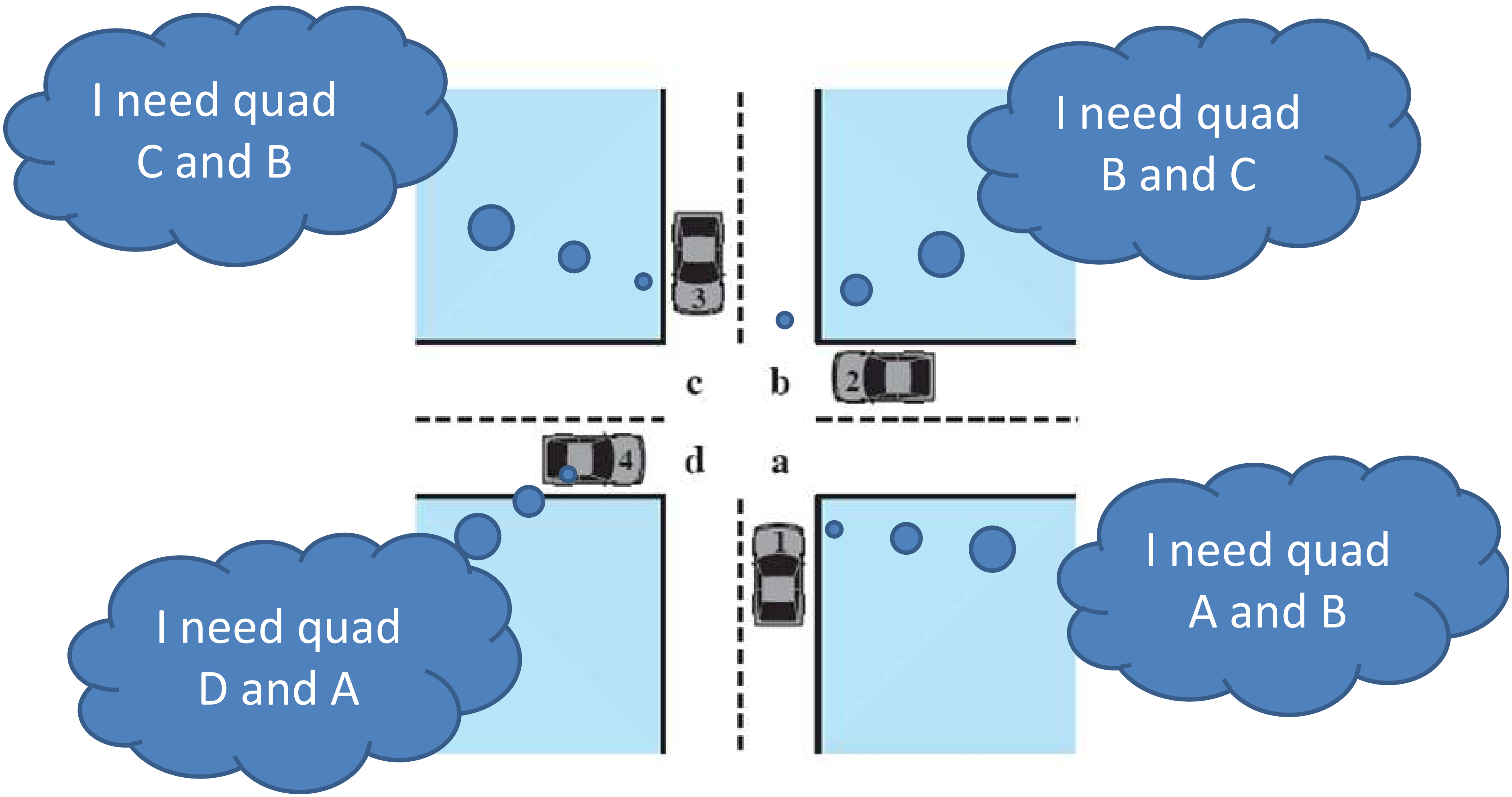
Barriers

- Synchronization mechanisms
- Intended for group of processes

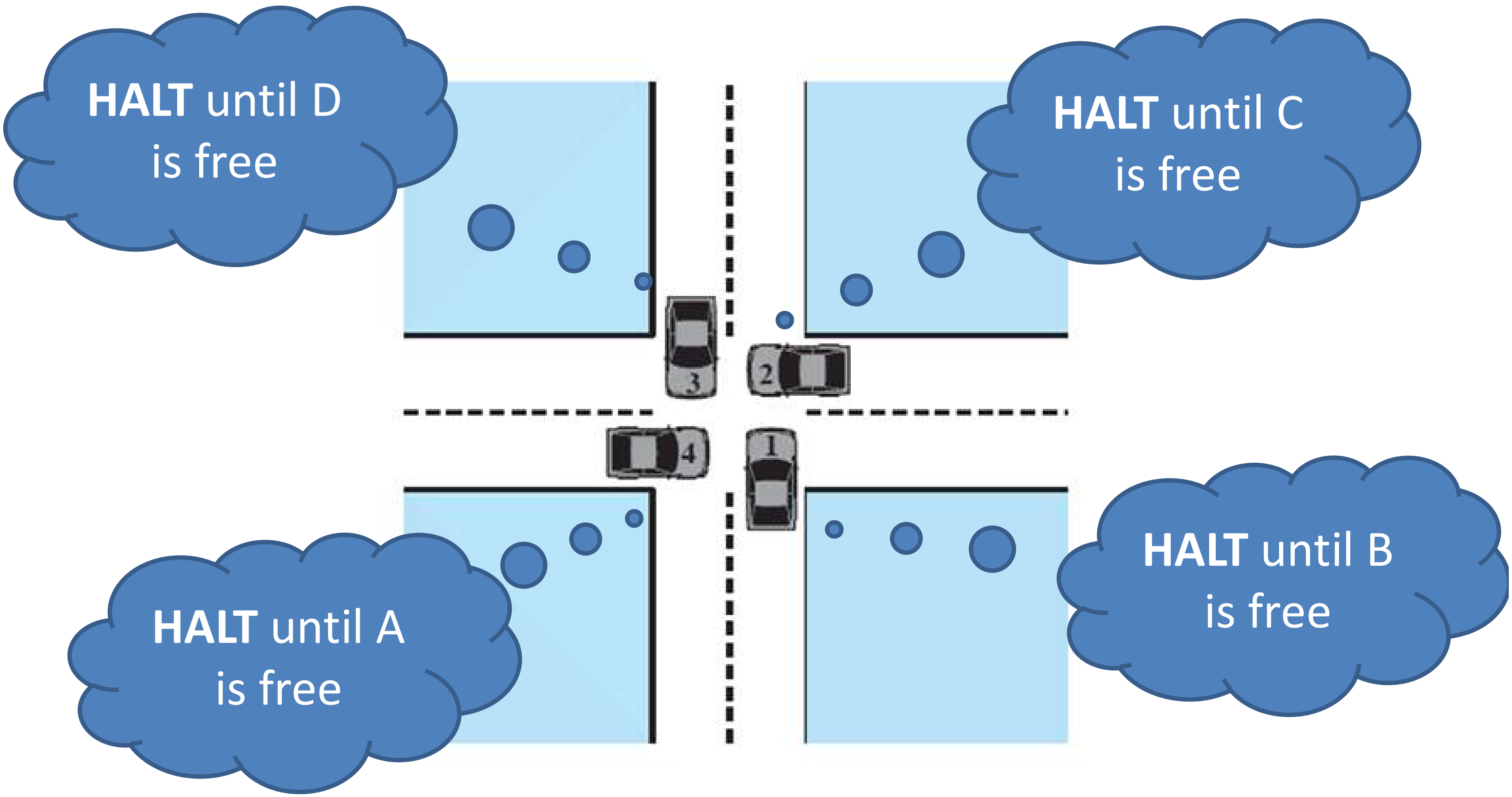



BANCO REAL
11455

Potential Deadlock



Actual Deadlock



Deadlocks

Occur among processes who need to acquire resources in order to progress

Resources

- Anything that must be acquired, used, and released over the course of time.
- Hardware or software resources
- Preemptable and Nonpreembtable resources:
  - Preemptable: can be taken away from the process with no ill-effect
  - Nonpreemptable: cannot be taken away from the process without causing the computation to fail

Resource Categories

Reusable

- can be safely used by only one process at a time and is not depleted by that use
  - processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Consumable

- one that can be created (produced) and destroyed (consumed)
  - interrupts, signals, messages, and information
  - in I/O buffers

```
typedef int semaphore;
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }
```

Deadlock-free code

```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }
```

Code with potential deadlock

```
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_2);
        down(&resource_1);
        use_both_resources( );
        up(&resource_1);
        up(&resource_2);
    }
```

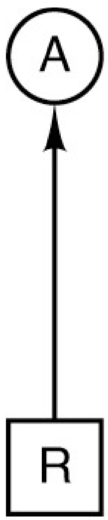
So …

- A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.
- Assumptions
  - If a process is denied a resource, it is put to sleep
  - Only single-thread processes
  - No interrupts possible to wake up a blocked process

Conditions for Resource Deadlocks

1. Each resource is either currently assigned to exactly one process or is available.
2. Processes currently holding resources that were granted earlier can request new resources.
3. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

Resource Allocation Graph

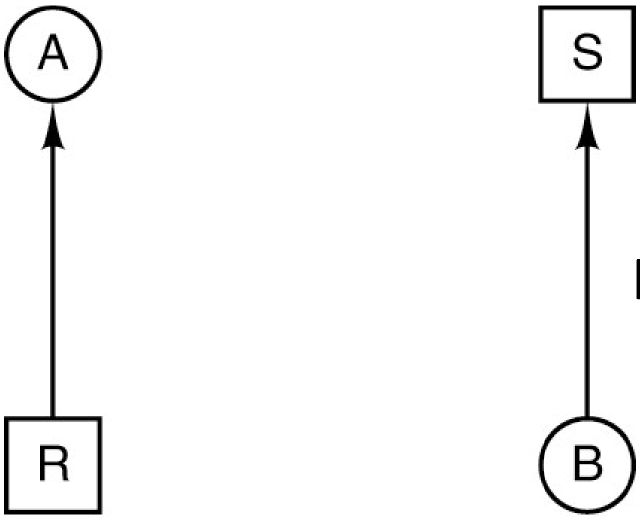


Process

Process A is holding resource R

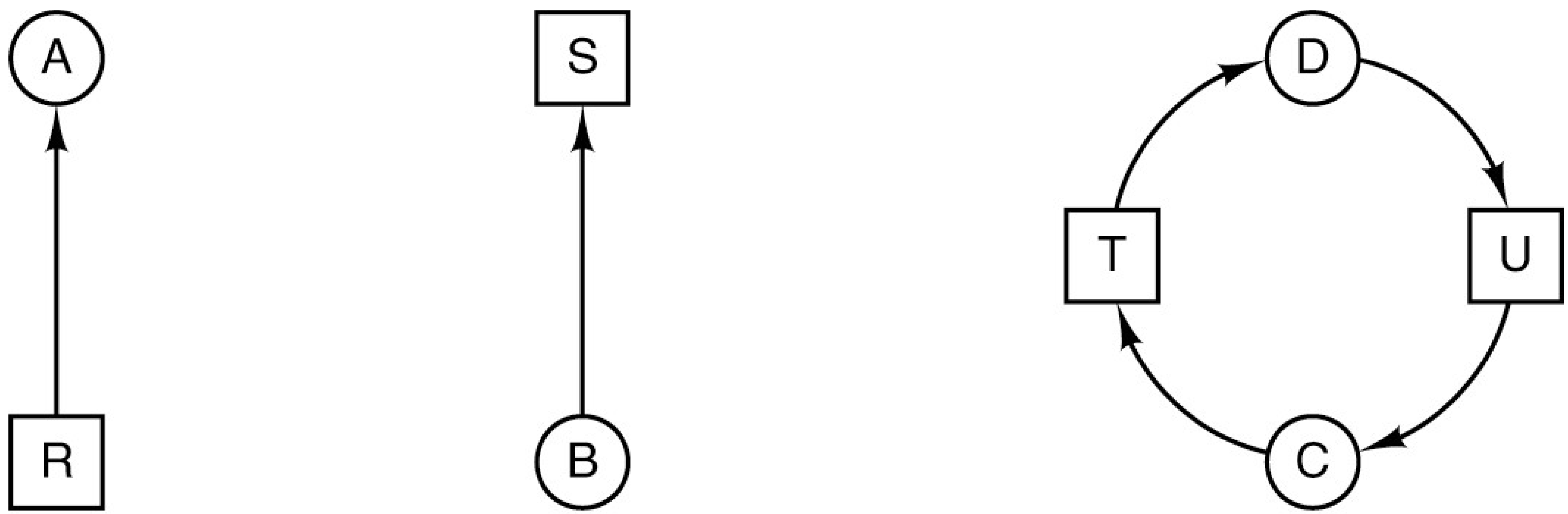
Resource

Resource Allocation Graph



Process B is requesting resource S

Resource Allocation Graph



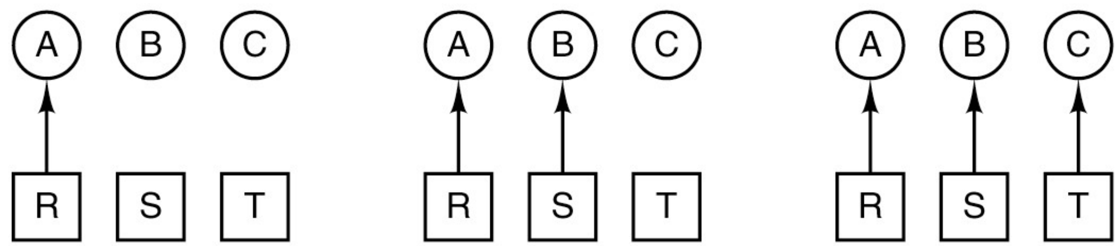
Deadlock!

Example 1:

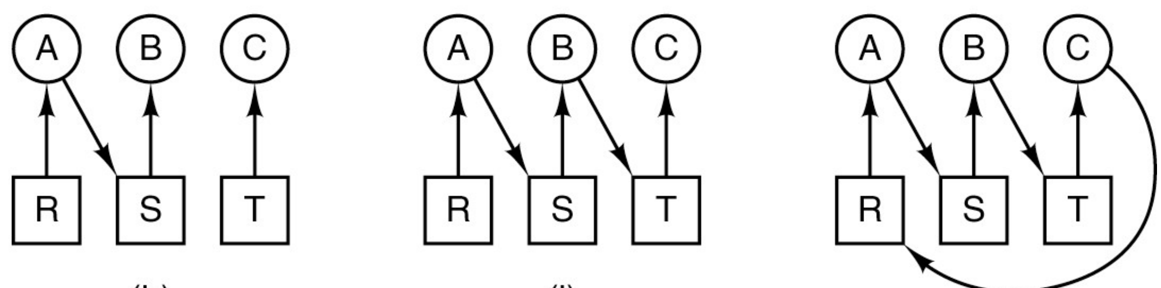
A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

(d)



(e) (f) (g)



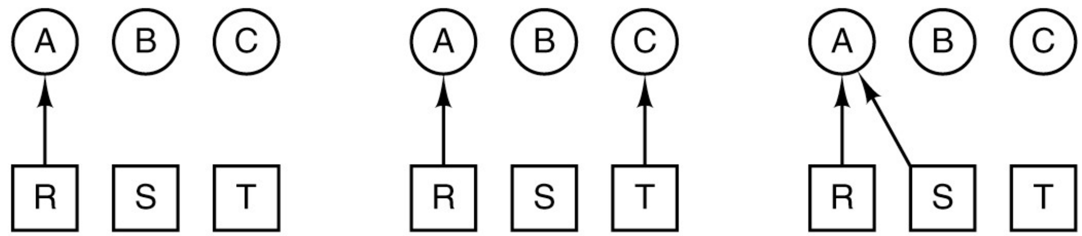
(h) (i) (j)

Example 2:

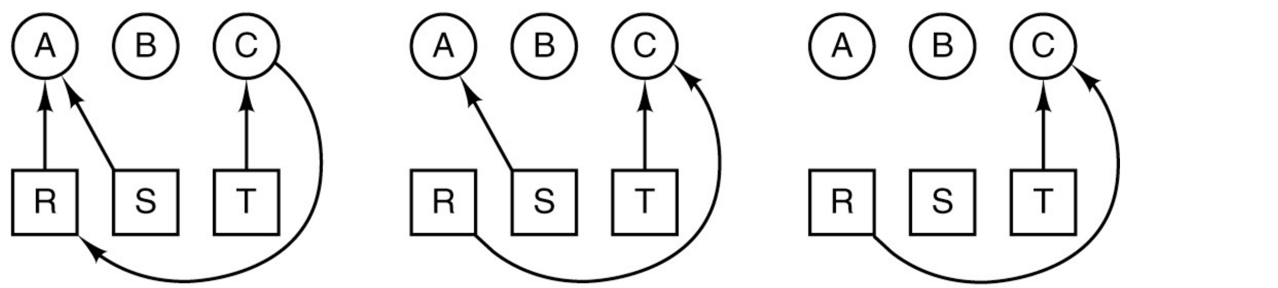
A	B	C
Request R	Request S	Request T
Request S	Request T	Request R
Release R	Release S	Release T
Release S	Release T	Release R

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
   no deadlock

(k)



(l) (m) (n)



(o) (p) (q)

How to Deal with Deadlocks

1. Just ignore the problem!
2. Let deadlocks occur, detect them, and take action
3. Dynamic avoidance by careful resource allocation
4. Prevention, by structurally negating one of the four required conditions (slide 45).

Just ignore the problem!

The Ostrich Algorithm

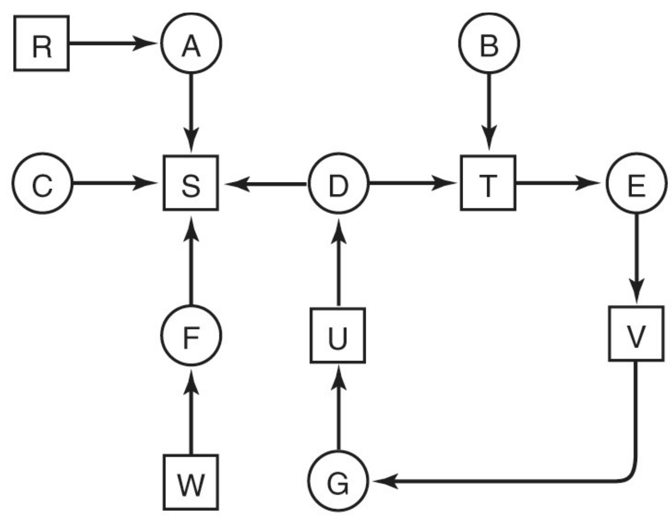
Let deadlocks occur, detect them, and take action

Deadlock Detection and Recovery

- The system does not attempt to prevent deadlocks.
- It tries to detect it when it happens.
- Then it takes some actions to recover
- Several issues here:
  - Deadlock detection with one resource of each type
  - Deadlock detection with multiple resources of each type
  - Recovery from deadlock

Deadlock Detection: One Resource of Each Type

- Construct a resource graph
- If it contains one or more cycles, a deadlock exists



Formal Algorithm to Detect Cycles in the Allocation Graph

For Each node N in the graph do:

1. Initialize L to empty list and designate all arcs as unmarked
2. Add the current node to end of L. If the node appears in L twice then we have a cycle and the algorithm terminates
3. From the given node pick any unmarked outgoing arc. If none is available go to 5.
4. Pick an outgoing arc at random and mark it. Then follow it to the new current node and go to 2.
5. If the node is the initial node then no cycles and the algorithm terminates. Otherwise, we are in dead end. Remove that node and go back to the previous one. Go to 2.

Deadlock Detection: Multiple Resources of Each Type

n processes and m resource types

Resources in existence
(E_1, E_2, E_3, É, E_m)

Resources available
(A_1, A_2, A_3, É, A_m)

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

A Process is said to be marked if they are able to complete and hence not deadlocked

Deadlock Detection: Multiple Resources of Each Type

n processes and m resource types

Resources in existence
$(E_1, E_2, E_3, \text{É}, E_m)$

Resources available
$(A_1, A_2, A_3, \text{É}, A_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Request matrix

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 needs

Steps of the deadlock detection algorithm:

1. Look for an unmarked process, Pi, for which the ith row of R <= A
2. If such process is found, Add ith row of C to A, mark the process, and go to step 1.
3. If no such process exists and there are unmarked processes → deadlock

When to Check for Deadlocks?

- Check every time a resource request is made
- Check every k minutes
- When CPU utilization has dropped below a threshold

Recovery from Deadlock

- We have detected a deadlock … What next?
- We have some options:
  - Recovery through preemption
  - Recovery through rollback
  - Recovery through killing processes

Recovery from Deadlock: Through Preemption

- Temporary take a resource away from its owner and give it to another process
- Manual intervention may be required (e.g. in case of printer)
- Highly dependent on the nature of the resource.
- Recovering this way is frequently impossible.

Recovery from Deadlock: Through Rollback

- Have processes checkpointed periodically
- Checkpoint of a process: its state is written to a file so that it can be restarted later
- In case of deadlock, a process that owns a needed resource is rolled back to the point before it acquired that resource

Recovery from Deadlock: Through Killing Processes

- Kill a process in the cycle.
- Can be repeated (i.e. kill other processes) till deadlock is resolved
- The victim can also be a process NOT in the cycle

Dynamic avoidance by careful resource allocation

Deadlock Avoidance

- In most systems, resources are requested one at a time.
- Resource is granted only if it is safe to do so

Safe and Unsafe States

- A state is said to be safe if there is one scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately
- An unsafe state is NOT a deadlock state

Safe and Unsafe States

Maximum the process will need (e.g. A will need 6 more).

(a)

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(b)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(c)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(d)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(e)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

Assume a total of 10 instances of the resources available
Therefore "Free: x" means we have x instances available.

This state is safe because there exists a sequence of allocations that allows all processes to complete.

Safe and Unsafe States

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0
(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4
(d)

How about this state?

The difference between a safe and unsafe state is that from a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.

The Banker's Algorithm

- Dijkstra 1965
- Checks if granting the request leads to an unsafe state.
- If it does, the request is denied.

The Banker's Algorithm: The Main Idea

- The algorithm checks to see if it has enough resources to satisfy some customers
- If so, the process closest to the limit is assumed to be done and resources are back, and so on.
- If all loans (resources) can eventually be repaid, the state is safe.

The Banker's Algorithm: Example (single resource type)

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

Safe

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

Safe

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

Unsafe

The Banker's Algorithm: Example (multiple resources)

Process	Tape drives	Plotters	Printers	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

The Banker's Algorithm

- Very nice theoretically
- Practically useless!
  - Processes rarely know in advance what their maximum resource needs will be.
  - The number of processes is not fixed.
  - Resources can suddenly vanish.

Prevention, by structurally negating one of the four required conditions (slide 45).

Deadlock Prevention

- Deadlock avoidance is essentially impossible.
- If we can ensure that at least one of the four conditions of the deadlock is never satisfied, then deadlocks will be structurally impossible.

Deadlock Prevention: Attacking the Mutual Exclusion

- Can be done for some resources (e.g the printer) but not all.
- Spooling: saves the data with the OS till the resource becomes available
  - e.g. A file to be printed is stored with the OS till the printer becomes available.

Deadlock Prevention:
Attacking the Hold and Wait Condition

- Prevent processes holding resources from waiting for more resources.
- This requires all processes to request all their resources before starting execution.
- A different strategy: require a process requesting a resource to first temporarily release all the resources it currently holds. Then tries to get everything it needs all at once

Deadlock Prevention: Attacking No Preemption Condition

- Virtualizing some resources can be a good strategy (e.g. virtualizing a printer)
- Not all resources can be virtualized (e.g. records in a database)

Deadlock Prevention: The circular Wait Condition

- Method 1: Have a rule saying that a process is entitled to only a single resource at a moment.
- Method 2:
  - Provide a global numbering of all resources.
  - A process can request resources whenever they want to, but all requests must be done in numerical order
  - With this rule, resource allocation graph can never have cycles.

Deadlock Prevention: Summary

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

Conclusions

- Race condition can occur when there are several processes and/or threads.
- Mutual exclusion deals with race condition but can cause deadlock.
- Deadlocks can occur on hardware/software resources
- OS need to be able to:
  - Detect deadlocks
  - Deal with them when detected
  - Try to avoid them if possible