Modern Operating Systems

Fifth Edition, Global Edition

Modern Operating Systems

FIFTH EDITION

Tanenbaum • Bos

Lecture 3

Hazards & Synchronization

Copyright © 2023 Pearson Education, Ltd. All Rights Reserved

Hazards: Processes & Threads

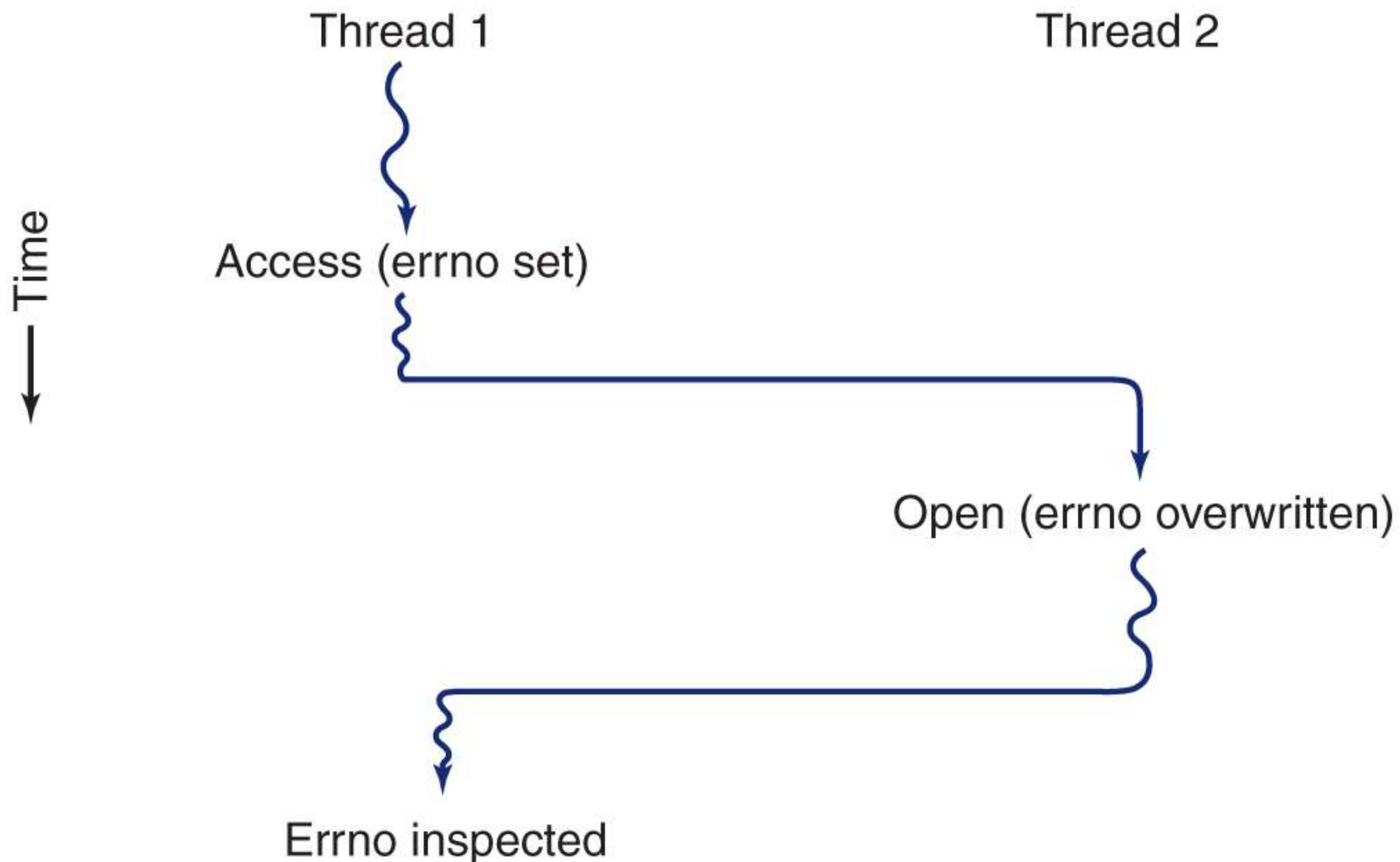
Threads can

- overwrite each others' stack
- change heap values
- access data structures in transient states
- access resources in unexpected interleaving
- …

Race Conditions

A race condition occurs when the ordering of execution between two processes (or threads) can affect the outcome of an execution (mostly unacceptable)

Conflicts between threads over the use of a global variable (or processes over the use of a shared variable)



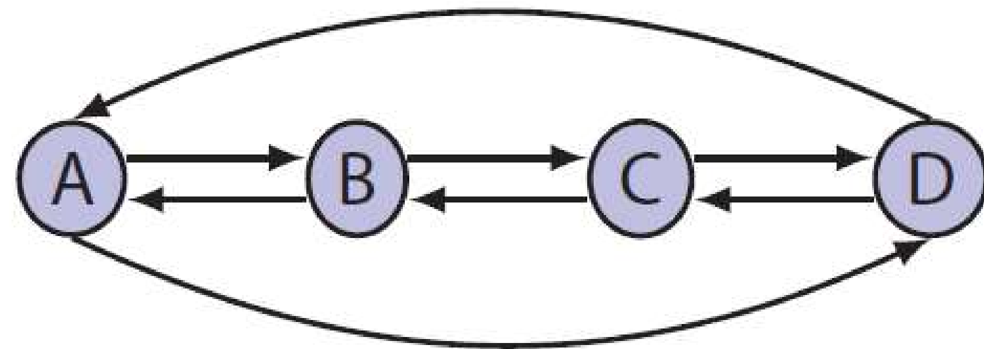
Race Conditions - Example

t	count++	count--
1	LOAD r, count	
2	ADD 1, r	
3		LOAD r, count
4		SUB 1, r
5	STORE r, count	
6		STORE r, count

Race condition
Due to interleaving, results depend on the order of memory accesses

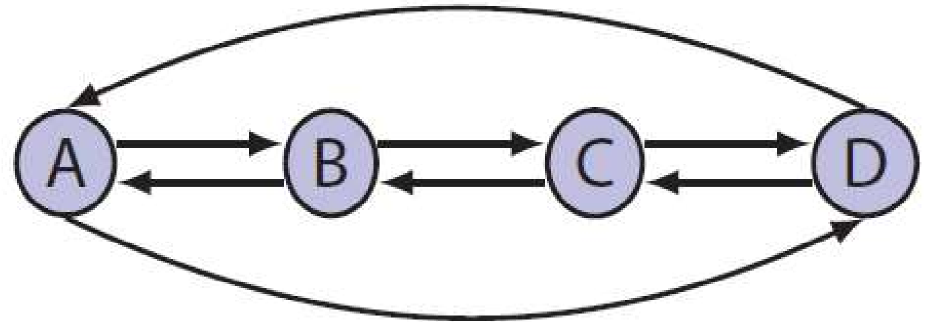
Race Conditions – Doubly Linked List

```
struct list_head {
    struct list_head *next;
    struct list_head *prev;
};

list_del(prev, next)
{
    prev->next = next;
    next->prev = prev;
}
```

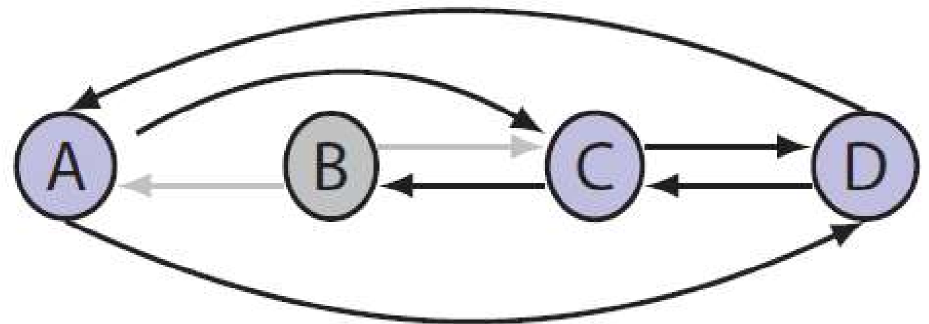


Race Conditions – Doubly Linked List

```
struct list_head {
    struct list_head *next;
    struct list_head *prev;
};
```

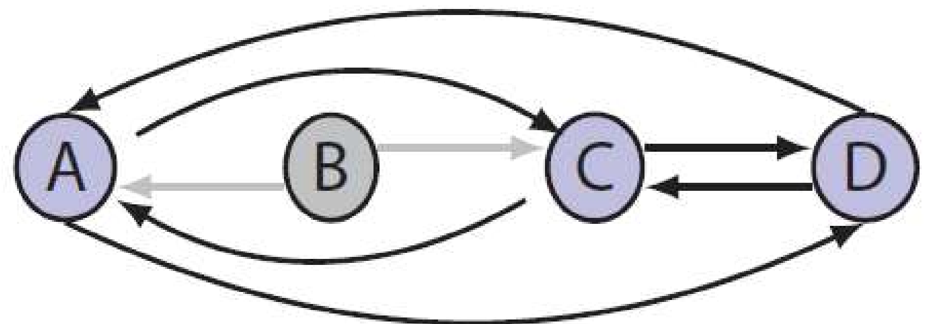



```
list_del(prev, next)
{
    prev->next = next;
    next->prev = prev;
}
```



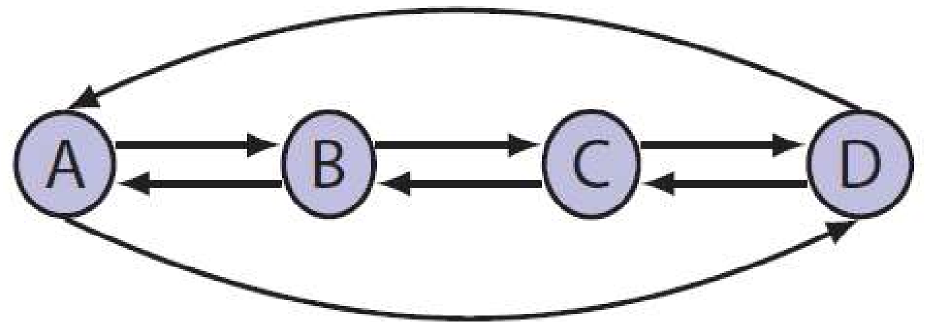
Διαγραφή B (1 διεργασία)

1. `t0: prev->next = next;`
2. `t0: next->prev = prev;`
3. OK!

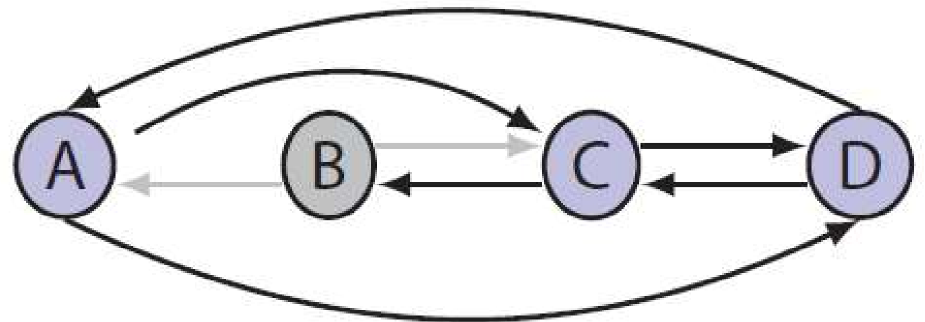


Race Conditions – Doubly Linked List

```
struct list_head {
    struct list_head *next;
    struct list_head *prev;
};
```

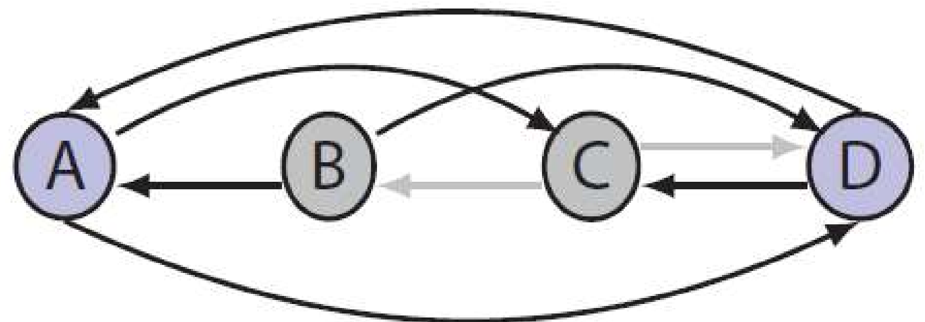



```
list_del(prev, next)
{
    prev->next = next;
    next->prev = prev;
}
```



Διαγραφή B,C (2 διεργασίες)

1. `t0: prev->next = next;`
2. `t1: prev->next = next;`

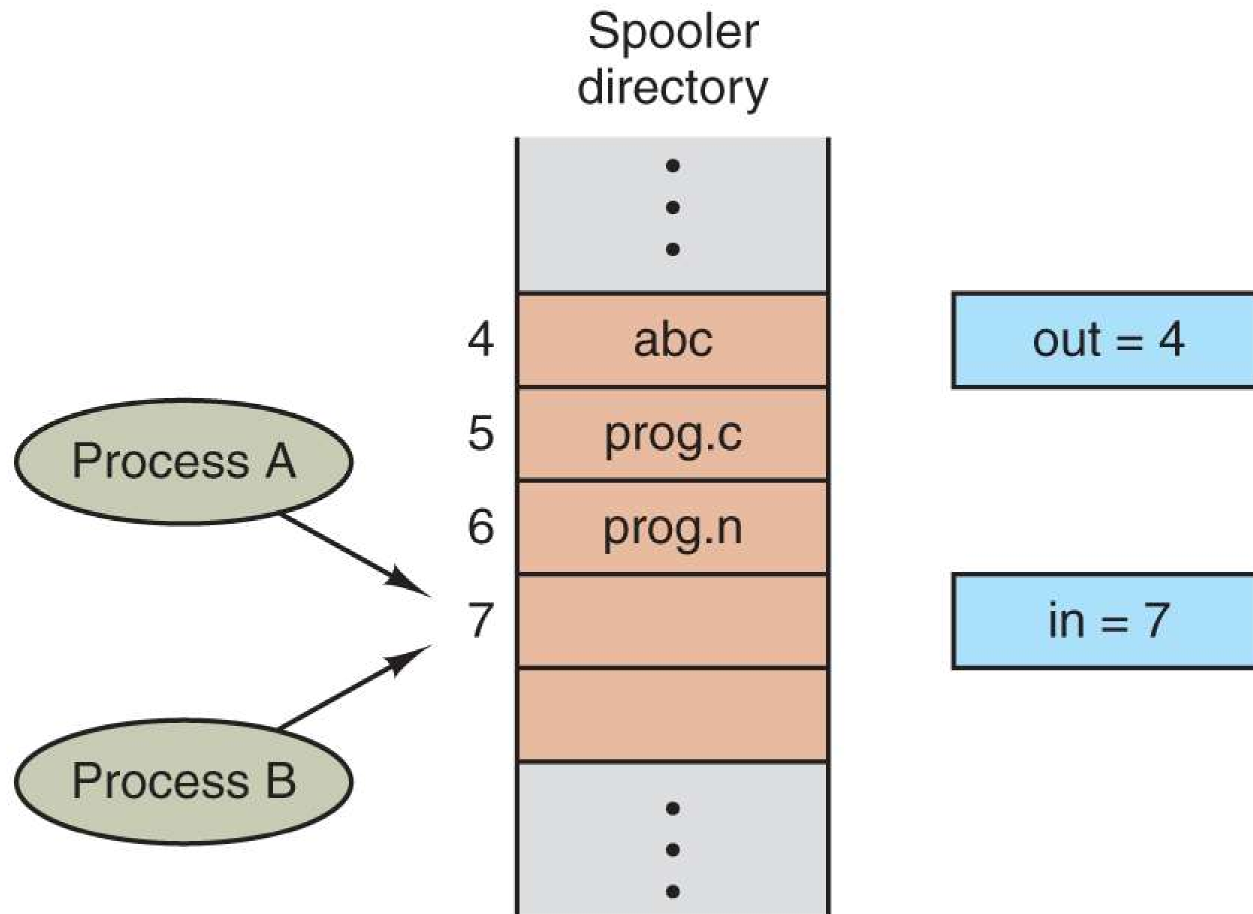


Inconsistency

Critical Section & Mutual Exclusion

Two (or more) processes access shared data at the same time to perform an operation (medial property)

It consists of *entry* (negotiation), *critical section/region*, and *exit*



Critical Section & Mutual Exclusion

P1	P2
// other code…	// other code…
entry();	entry();
x++;	x--;
exit();	exit();
// other code…	// other code…

Critical Section & Mutual Exclusion

Critical regions must meet the following 3 conditions:

- **Mutual exclusion** – No more than one process can access the shared data in the critical section
- **Progress** – If no process accesses the shared data, then:
  - Only processes executing the entry/exit sections can affect the selection of the next process to enter the critical section
  - The process of selection must eventually complete
- **Bounded waiting** – Once a process executes its entry section, there is an upper bound on the number of times that other processes can enter the region of mutual exclusion

Critical Section – Locks

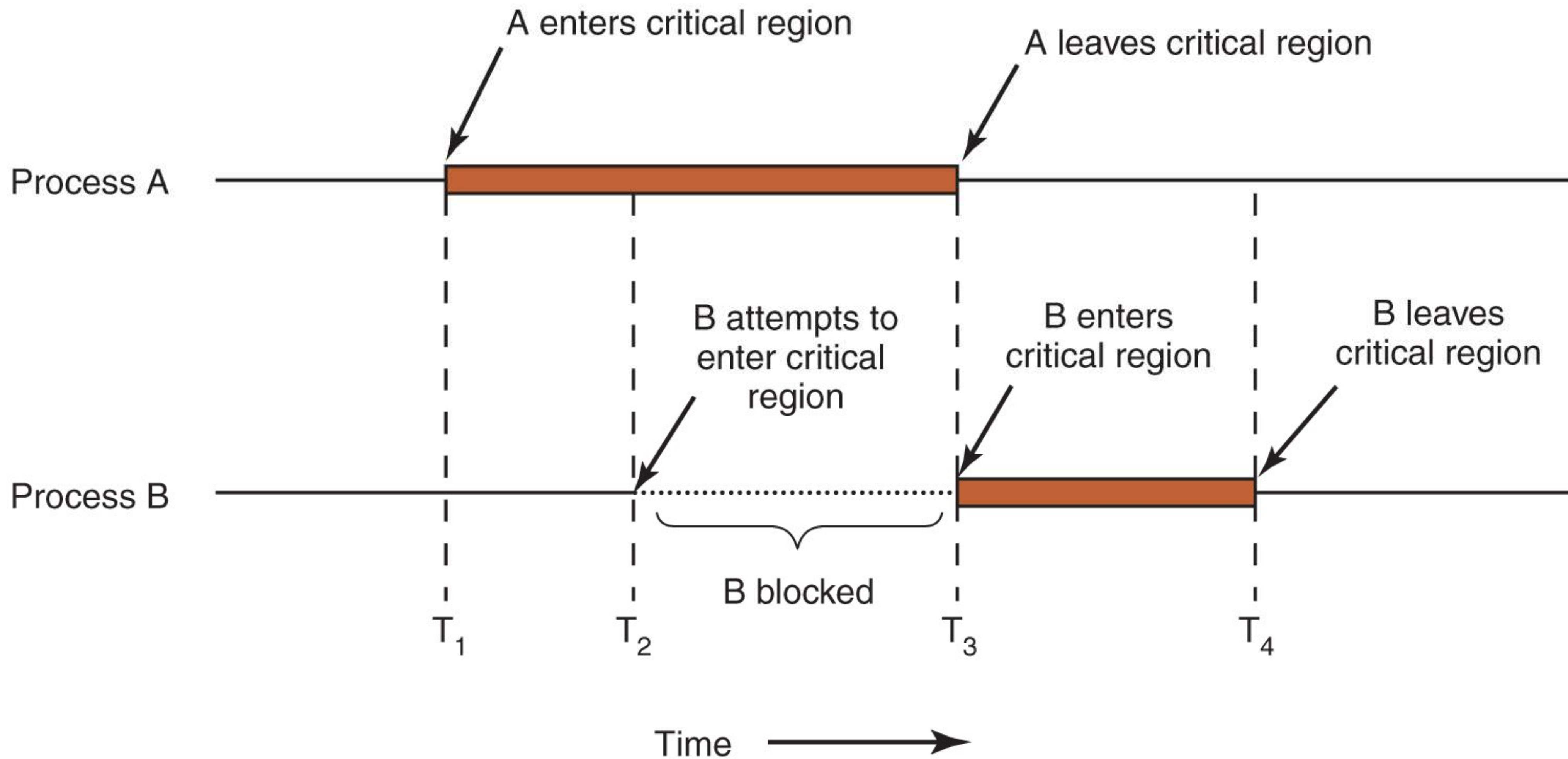
- software only
- hardware/software
- abstractions of the critical region problem
  - data types
  - language constructs

```
do {
    lock(mylock);
        /* critical section */
        code
    unlock(mylock);
    remainder section
} while (TRUE);
```

```
do {
    lock(mylock);
        /* critical section */
        other code
    unlock(mylock);
    other remainder section
} while (TRUE);
```

Mutual Exclusion

Mutual exclusion using critical sections



A Solution to Critical Section Problem

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

(a) Process 0

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

(b) Process 1

strictly alternating

```
bool flag[2] = { FALSE, FALSE };
do {
    flag[me] = TRUE;
    while (flag[other])
        ;
        critical section
    flag[me] = FALSE;
    remainder section
} while (TRUE);
```

t	me=0	me=1
1	flag[0] = TRUE	
2		flag[1] = TRUE

deadlock

Peterson's Solution

```
do {
    flag[me] = TRUE;
    turn = other;
    while (flag[other] && turn == other)
        ;
        critical section
    flag[me] = FALSE;
    remainder section
} while (TRUE);
```

Theoretical, but not so practical (N=2, instruction reorder)

Bakery algorithm provides solutions for >2 processes

Peterson solution for mutual exclusion

```
#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 */
}
```

Bakery algorithm provide solutions for >2 processes

Bakery Algorithm: The Idea

- Each process “takes a ticket number” before entering the critical section.
- The ticket number indicates the order of entry.
- The process with the smallest ticket number enters first.
- If there is a tie, the process with the smaller ID goes first.
- After leaving the critical section, the process resets its ticket number to zero (indicating it no longer wants to enter).

Lamport’s Bakery Algorithm

```
bool choosing[N];  // choosing[i] = TRUE means process i is picking a number
int number[N];     // number[i] is the ticket number for process I
choosing[i] = TRUE; // Before entering critical section
// Pick the highest ticket number among all processes + 1
number[i] = 1 + max(number[0], number[1], ..., number[N-1]);
choosing[i] = FALSE; // Wait until it's my turn
for (int j = 0; j < N; j++) {
    while (choosing[j]) { /* busy wait while process j is choosing its number*/ }
    // Wait if process j has a smaller ticket or the same number but smaller ID
    while (number[j] != 0 &&
           (number[j] < number[i] ||
            (number[j] == number[i] && j < i))) {
        /* busy wait */
    }
}
// CS
// After critical section
number[i] = 0;
```

Atomic Ops: Test&Set (Enter/Leave CS)

```
bool TestAndSetLock(bool *Target) {
        bool Result;
        Result = *Target
        *Target = true;
        return Result;
}
```

instructions cannot be interrupted

```
shared bool PreventEntry = false;
        repeat // entry
                while TestAndSetLock(&PreventEntry) no-op;
                // CS
                PreventEntry = false;  //exit
        until NoLongerNeeded();
```

Test & Set (Enter/Leave a CS)

```
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
```

Spinlock using Atomics - Busy-Wait

```
do {
  while (TestAndSet(lock))
     ;
        critical section
  UnSet(lock);
  remainder section
} while (TRUE);
```

```
do {
  key = TRUE;
  while (key)
     Swap(lock,key);
        critical section
  UnSet(lock);
  remainder section
} while (TRUE);
```

Solution does not provide bounded wait!

Ticker Lock (Bounded Wait)

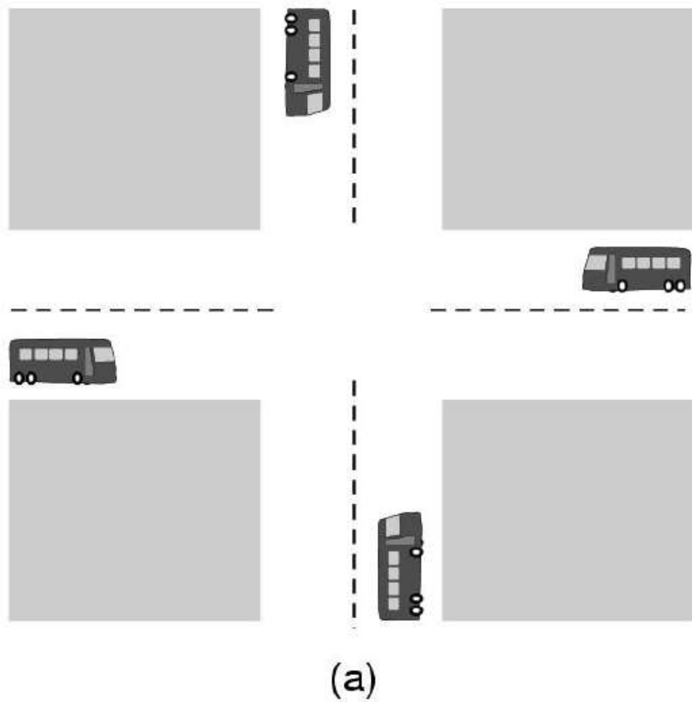
```
#define NEXT(x) ((x + 1) % N)
bool waiting[N], lock;

Lock:
    Set(waiting[me]);
    key = TRUE;
    while (Test(waiting[me]) && key)
        key = TestAndSet(&lock);
    UnSet(waiting[me]);

Unlock:
    for (j = NEXT(me); ; j = NEXT(j)){
        if (j == me){
            UnSet(lock); break;
        }
        if (Test(waiting[j])){
            UnSet(waiting[j]); break;
        }
    }
```

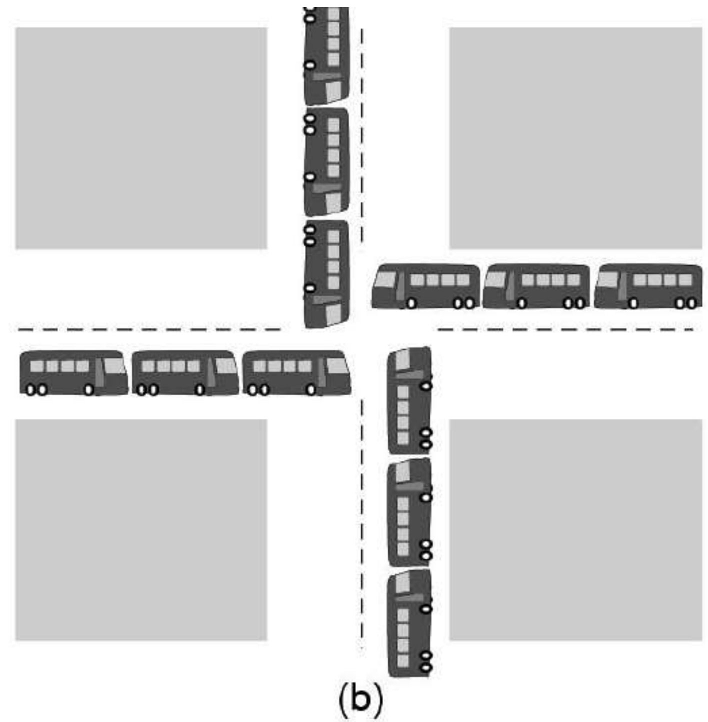
Compare & Swap (Enter/Leave a CS)

Entering and leaving a critical region using the XCHG instruction.

```
enter_region:
    MOVE REGISTER,#1            | put a 1 in the register
    XCHG REGISTER,LOCK          | swap contents of 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
```

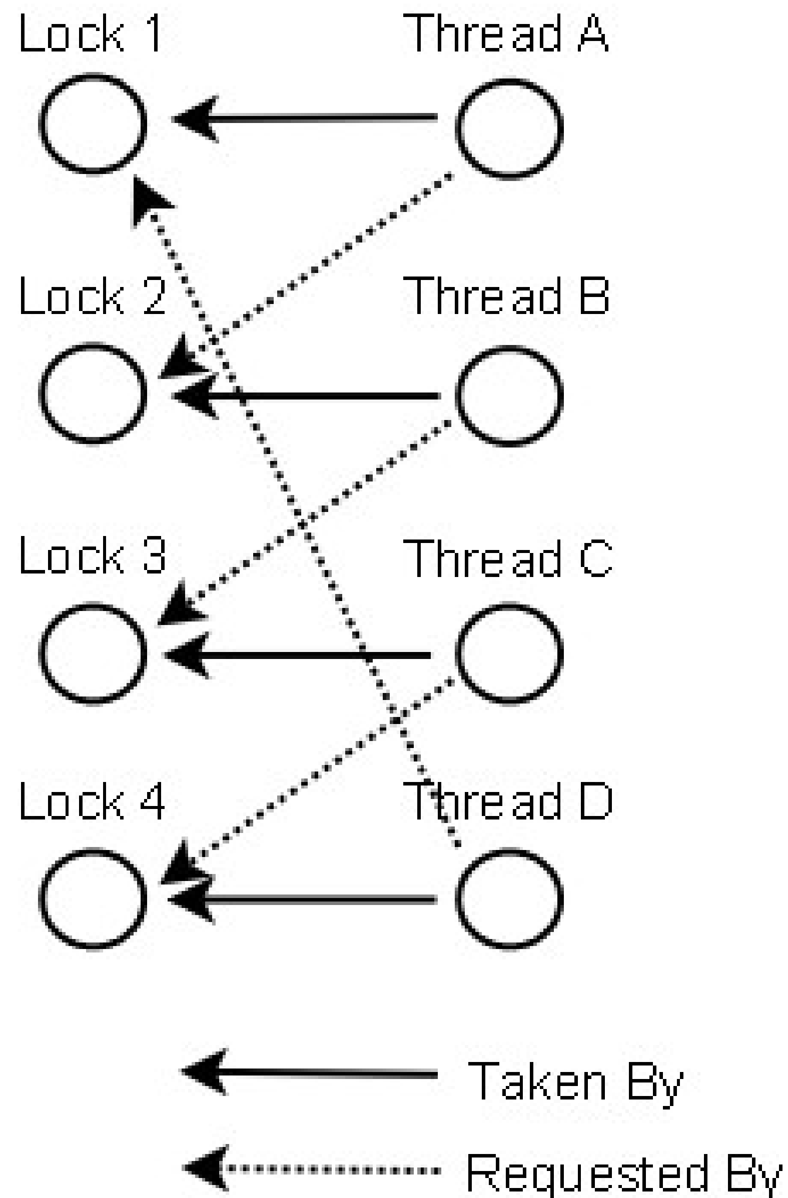
Process Deadlock


(a)


(b)

(a) A potential deadlock. (b) an actual deadlock

Process Deadlock



Necessary Conditions for Deadlock

Mutual Exclusion

At least one resource must not support a shared mode of operation (i.e., it cannot be used by more than one process at the same time).

Hold and Wait

While acquiring the resources, processes must:

- Hold resources they have already acquired
- Wait for additional resources to become available

No Preemption

- Resources can only be released voluntarily by the processes that hold them. They cannot be forcibly taken away.

Circular Wait

A set of processes exists such that: P_1 waits for a resource held by P_2, P_2 waits for a resource held by P_3 … P_{n-1}waits for a resource held by P_n, and $P_n$waits for a resource held by P_1

Preventing Deadlock

Not practical to negate:

- mutual exclusion
- no preemption

Leaves us with 2 remaining conditions to consider:

- Hold & wait
- No circular wait

Negating hold and wait

All resources must be requested at the same time.

If we need resources dynamically… each time we need a new resource:

- Release all held resources
- Acquire new set that is needed

Negating circular wait

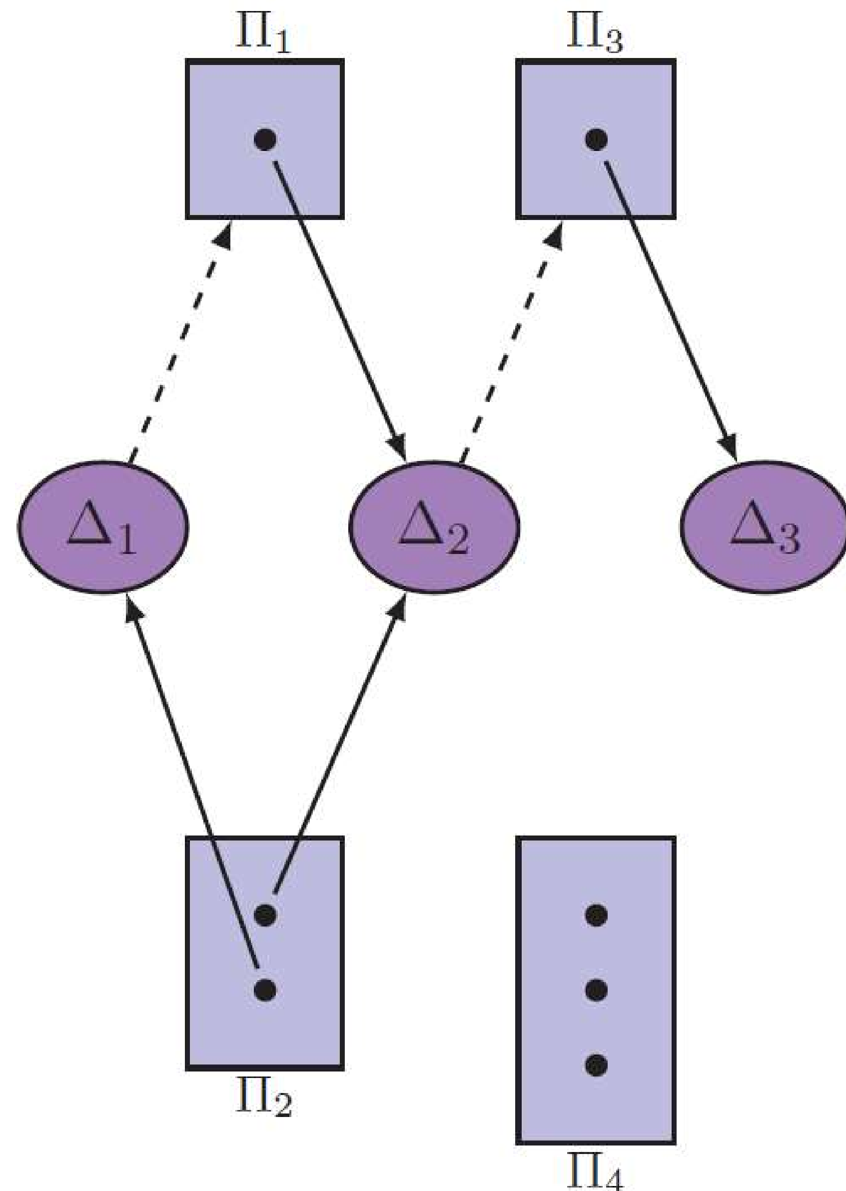
An ordering is defined on resources (e.g., they are numbered)

- If a process needs resource 1, 7, and 9, they must be acquired in that order.
- If the process later needs resource 8, it must release 9 before acquiring 8.

Breaks circular wait, but makes it hard to write portable code

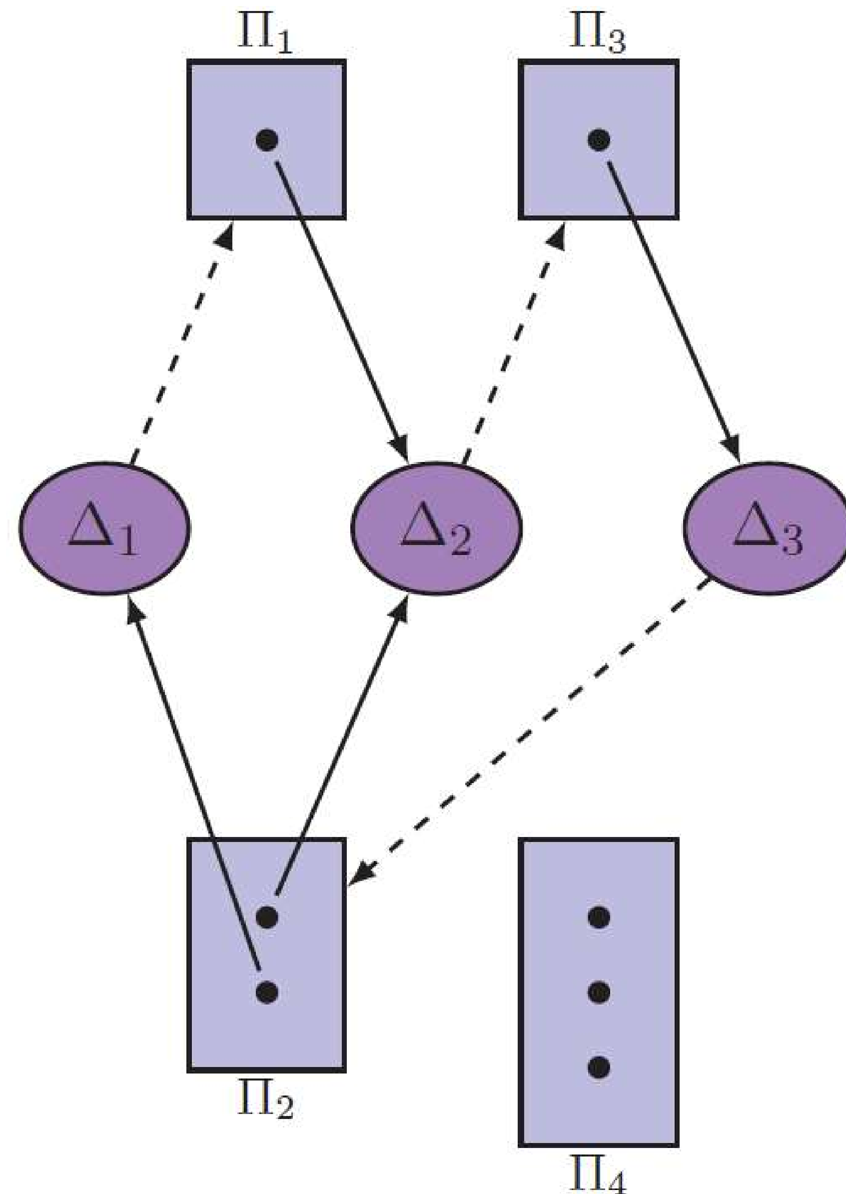
Resource Allocation Graph

- Resources Πx (instances)
- Processes Δx
- Request edge: Δx → Πx
- Assignment edge: Πx → Δx
- Refers to a specific moment in time of the system
- If there is no cycle, there is no
- If there is a cycle, then there may be a



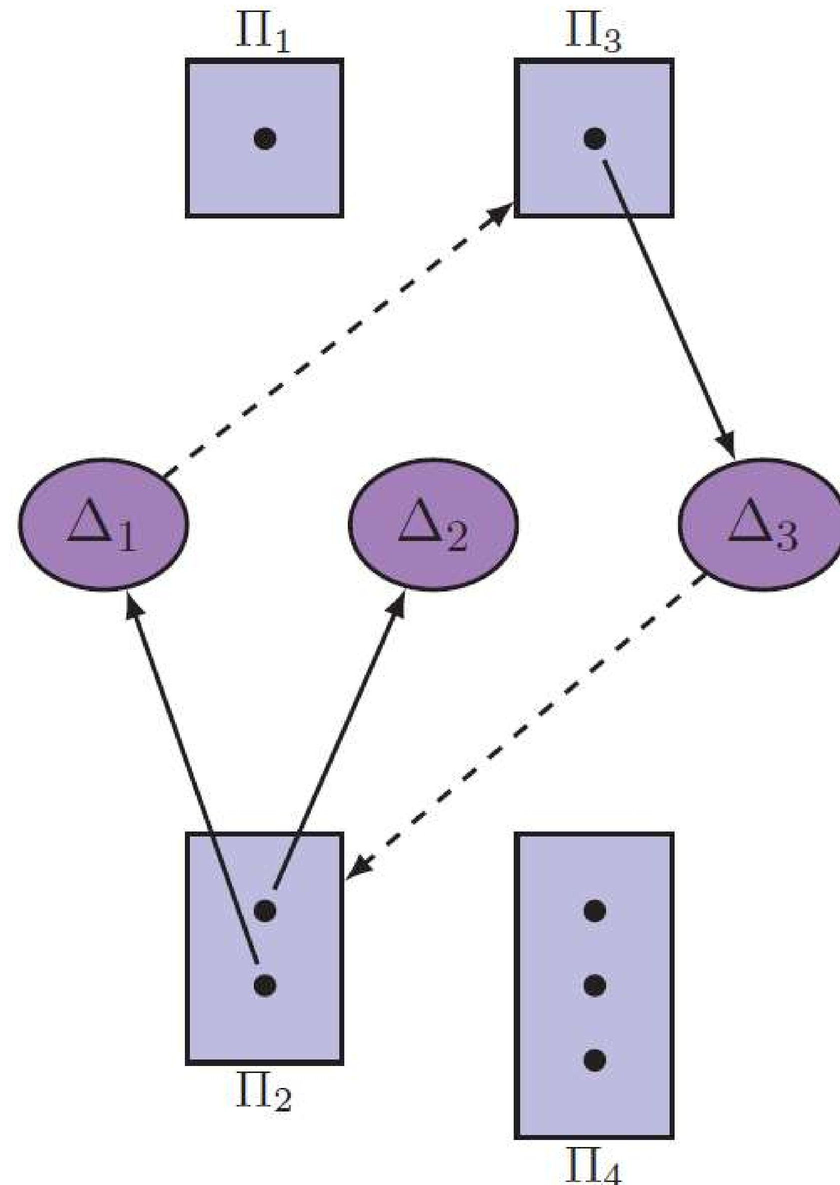
Resource Allocation Graph

- Resources Πx (instances)
- Processes Δx
- Request edge: Δx → Πx
- Assignment edge: Πx → Δx
- Refers to a specific moment in time of the system
- If there is no cycle, there is no
- If there is a cycle, then there may be a



Resource Allocation Graph

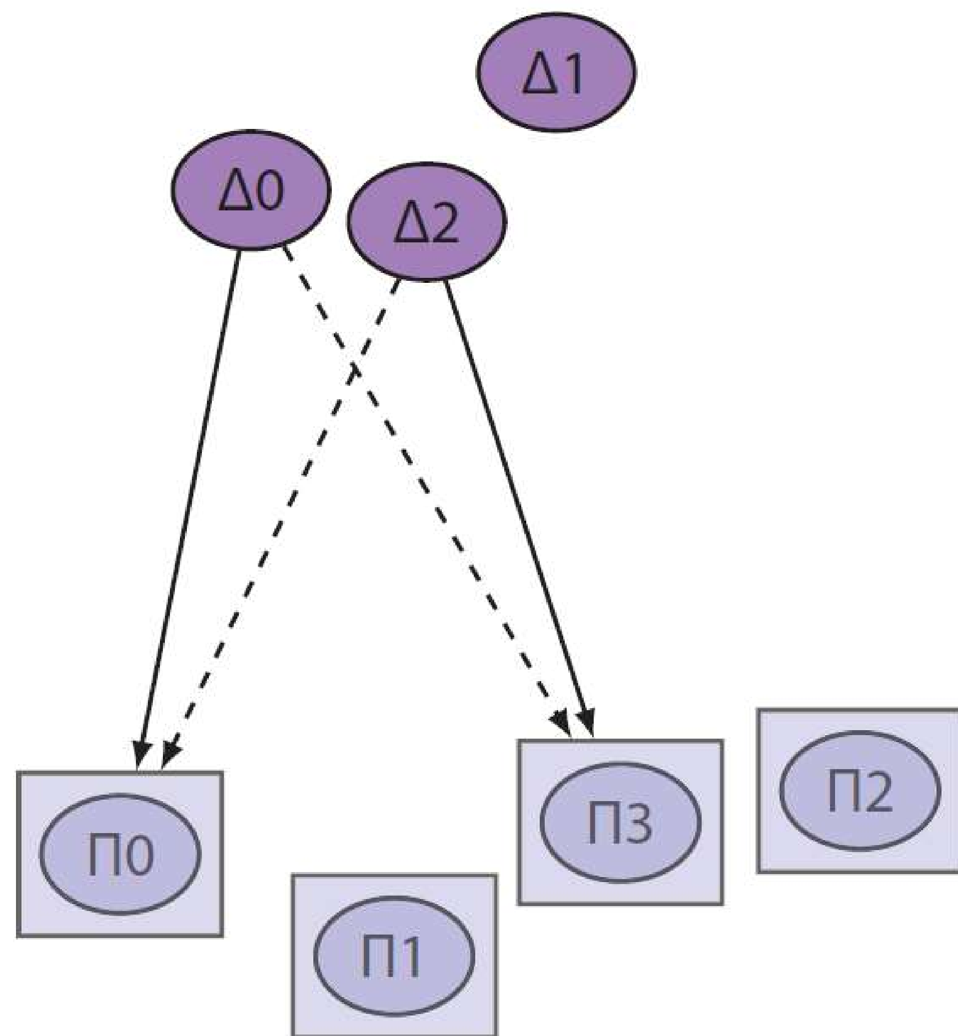
- Resources Πx (instances)
- Processes Δx
- Request edge: Δx → Πx
- Assignment edge: Πx → Δx
- Refers to a specific moment in time of the system
- If there is no cycle, there is no
- If there is a cycle, then there may be a



Central vs Fine-Grain Locking

Central Lock

- Any process that acquires the central lock can use **all resources**
- Simple design
- No s between resources
- Low parallelism (only one process effectively runs in the critical section)
- **Multiple Locks (Fine-Grain Locking)**
- One lock per available resource
- ✓ Enables parallelism
- ✗ Risk of . Example:
- Δ0 holds P0 and waits for P3
- Δ2 holds P3 and waits for P0



Transactional Memory

Transactional memory paradigms simplify synchronization through user-level atomic transactions that update multiple memory locations as a single operation, while avoiding locks

Transactional memory is alike database transactions

It is set by the developer. It can:

- Succeed and commit (COMMIT)
- Fail and be canceled (ABORT, ROLLBACK)

It can use hardware support with shadow memory

Semaphores (Non-Binary Locks)

- Previous solutions to Critical Section have relied on Locks
- A **semaphore** is an abstract data type for synchronization
- It contains an integer variable which is accessed by two operations known by many names:

P (test "prohoben")	wait	down*
V (increment "verhogen")	signal	up

	POSIX	Windows
down	sem_wait	WaitForSingleObject
up	sem_post	ReleaseSemaphore

Semaphore Operations (Dijskstra 1965)

- Each semaphore takes an initial integer value >=0
- **semaphore s = 1;** // for example, initialize to 1
- It is important to always initialize semaphores
- **wait** – Decrement the counter value. If the counter is less than zero, block the process
- **signal** – Increment the counter value. If processes have blocked on the semaphore, unblock one of the processes
- A process calling **wait()** cannot know the value of the semaphore and therefore if it will wait or not
- No rule about which thread among those waiting will continue its operation after a **signal()** call.

Semaphore Implementations

- Higher-level sync primitives
- More flexible than locks (more complex problems)

```
wait (S) {
 while (S <= 0)
     ;
 S--;
}
```

```
signal (S) {
 S++;
}
```

Semaphore Implementation (busy wait)

```
wait (S) {
  while (1) {
     lock(S_lock);
     if (S > 0) {
        S--;
        unlock(S_lock);
        break;
     }
     unlock(S_lock);
  }
}
```

```
signal (S) {
  lock(S_lock);
  S++;
  unlock(S_lock);
}
```

Semaphore Implementation- no busy wait

```
typedef struct {
  int value;
  lock s_lock;
  struct process *list;
} Semaphore;
```

```
wait (Semaphore *S) {
  lock(S->s_lock);
  S->value--;
  if (S->value < 0) {
     add proc to S->list;
     unlock(S->s_lock);
     block();
  }
  unlock(S->s_lock);
}
```

```
signal (Semaphore *S) {
  lock(S->s_lock);
  S->value++;
  if (S->value <= 0) {
     remove P from S->list;
     wakeup(P);
  }
  unlock(S->s_lock);
}
```

Semaphores - Common CS Code

```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
        down(&resource_1);
        use_resource_1( );
        up(&resource_1);
}
```

(a)

```
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);
}
```

(b)

Using a semaphore to protect resources. (a) One resource. (b) Two resources

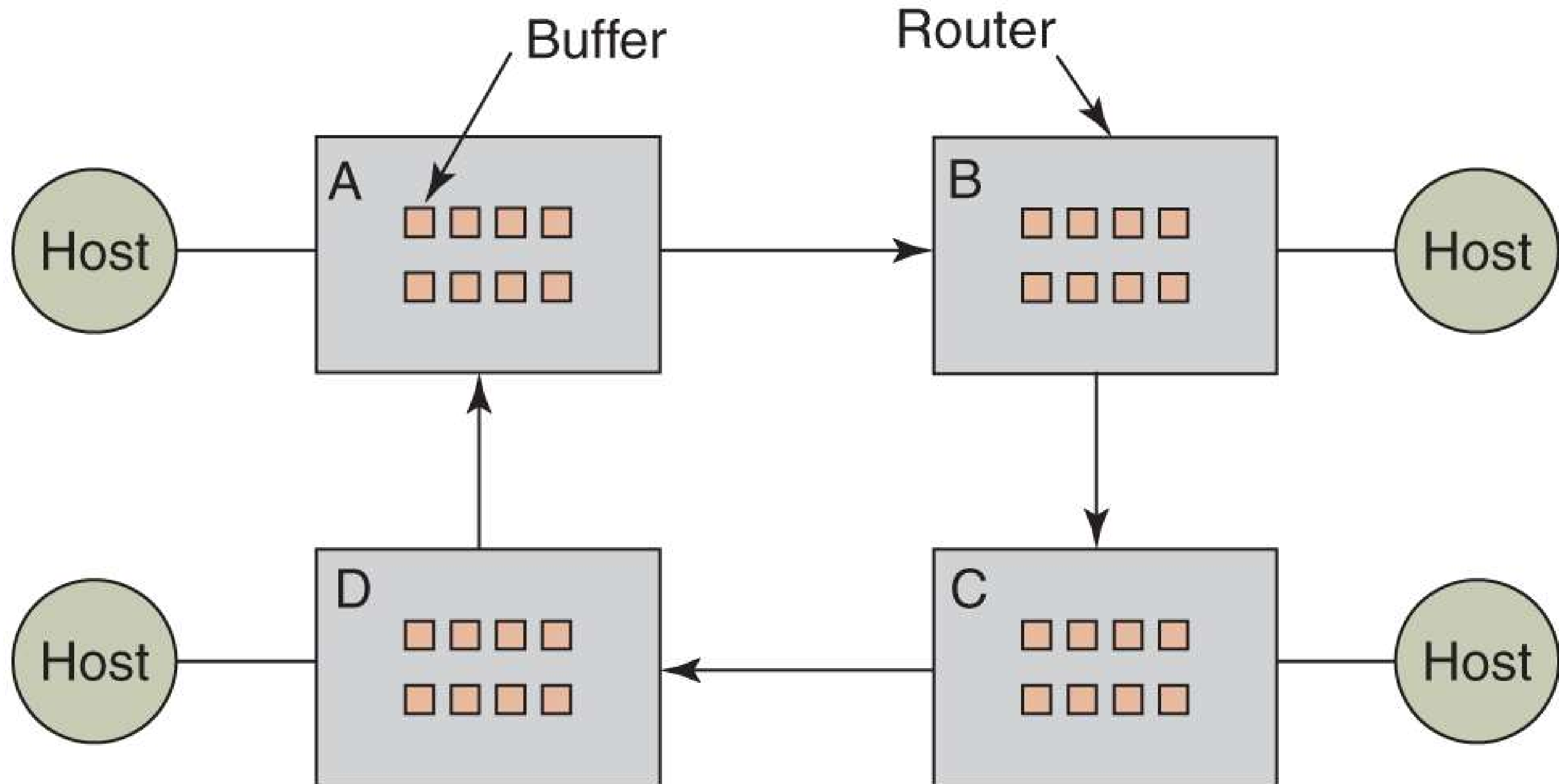
Semaphores - Further Examples

```
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);
    }
```

```
    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);
    }
```

(a) Deadlock-free code. (b) Code with a potential deadlock

Deadlock in a Network (Full Buffers)



Trylock & Livelock

```
void process_A(void) {
        acquire_lock(&resource_1);
        while (try_lock(&resource_2) == FAIL) {
                release_lock(&resource_1);
                wait_fixed_time();
                acquire_lock(&resource_1);
        }
        use_both_resources( );
        release_lock(&resource_2);
        release_lock(&resource_1);
}

void process_B(void) {
        acquire_lock(&resource_2);
        while (try_lock(&resource_1) == FAIL) {
                release_lock(&resource_2);
                wait_fixed_time();
                acquire_lock(&resource_2);
        }
        use_both_resources( );
        release_lock(&resource_1);
        release_lock(&resource_2);
}
```

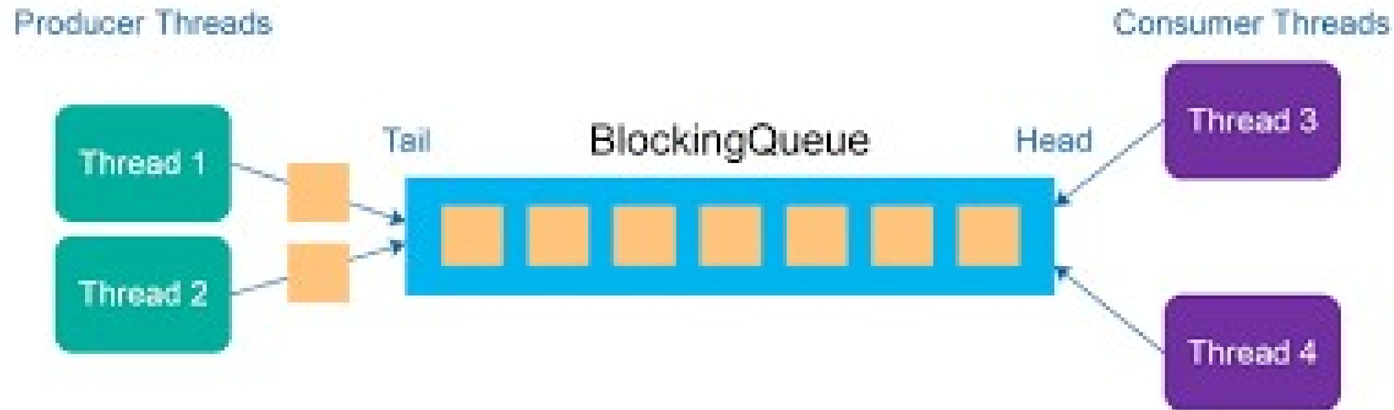
Ordering Example

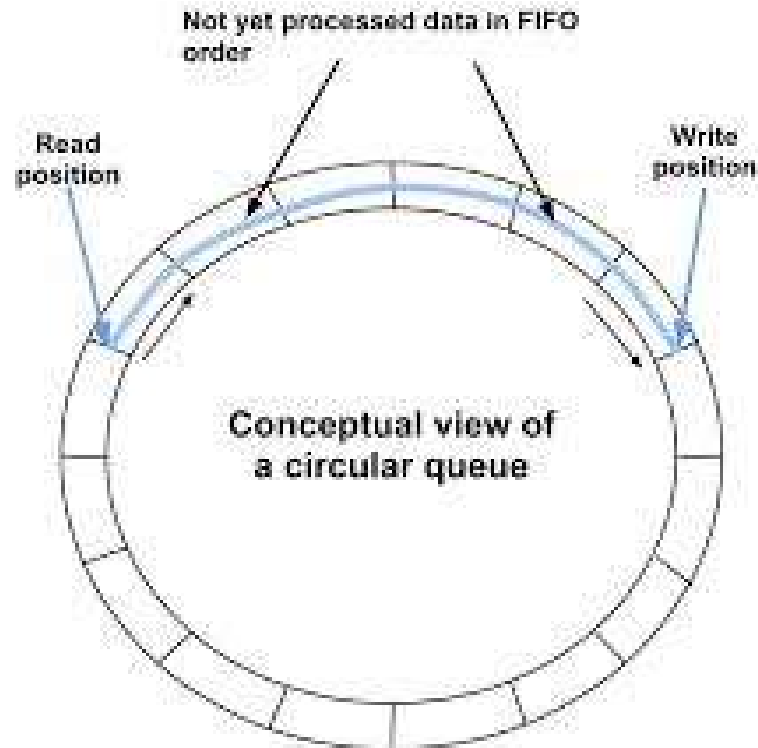
- We want process P_1 to execute routine S_1 before process P2 executes routine S2
- We initialize the semaphore to the value 0

```
// P1
S1();
signal(sema);
```

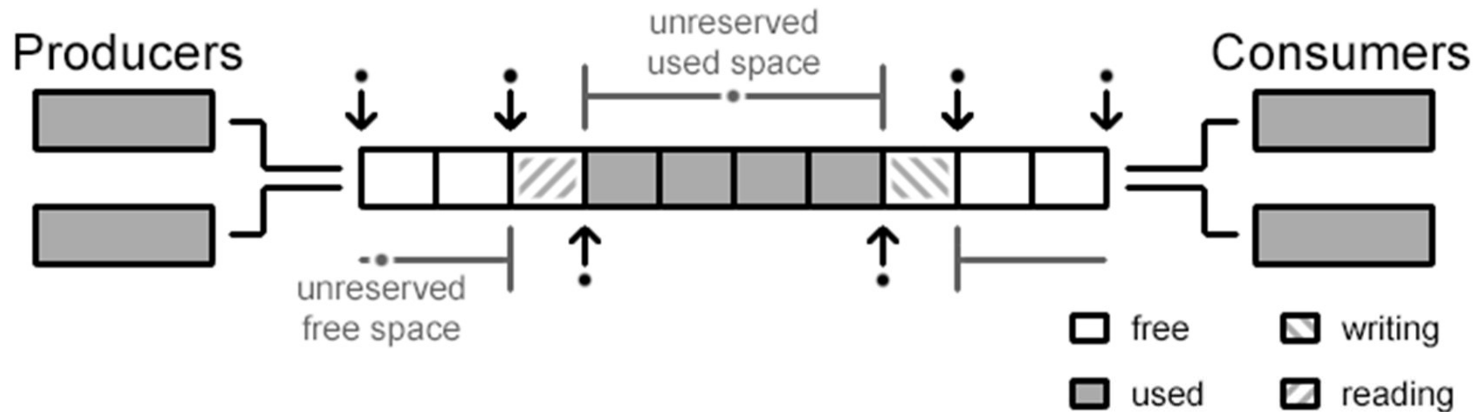
```
// P2
wait(sema);
S2();
```

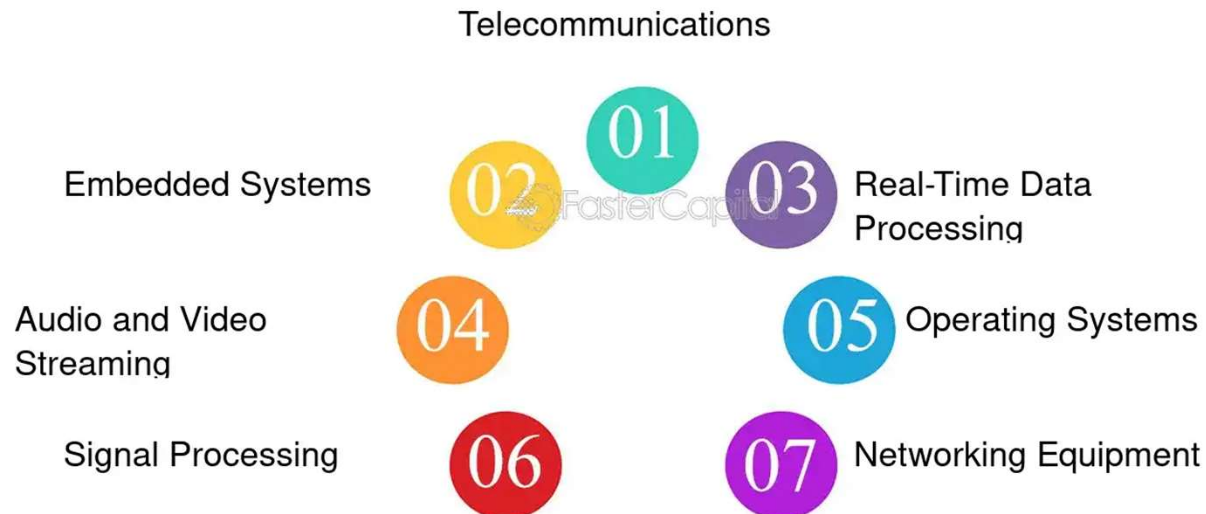
Producer-Consumer





Producer-Consumer (Deadlock)





Producer-Consumer

The producer-consumer problem with a fatal race condition.

```
#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 from buffer */
                count = count − 1;              /* decrement count of items in buffer */
                if (count == N − 1) wakeup(producer);   /* was buffer full? */
                consume_item(item);             /* print item */
        }
}
```

race condition?

Producer-Consumer (Deadlock)

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
sema_t mutex = semaphore(1);
```

```
void enqueue(item_t item){
    wait(mutex);
    item_t item;
    while (count == N)
        ; // wait
    buffer[in] = item;
    in = NEXT(in);
    count++;
    signal(mutex);
}
```

```
item_t dequeue(void){
    wait(mutex);
    item_t item;
    while (count == 0)
        ; // wait
    item = buffer[out];
    out = NEXT(out);
    count--;
    signal(mutex);
    return item
}
```

deadlock?

Producer-Consumer with semaphores

```
#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 */
     }
}
```

Monitors - Condition Variables

Mechanism for a process to suspend its operation until a condition is satisfied (e.g., Producer-Consumer)

Functions:

- wait(): The process becomes idle until signal() is called. Next
- signal(): wakes up (all, or exactly one) dormant process/es (if any exists)
- no init, value is always initialized to 0
- Only one active process in the monitor at any given time

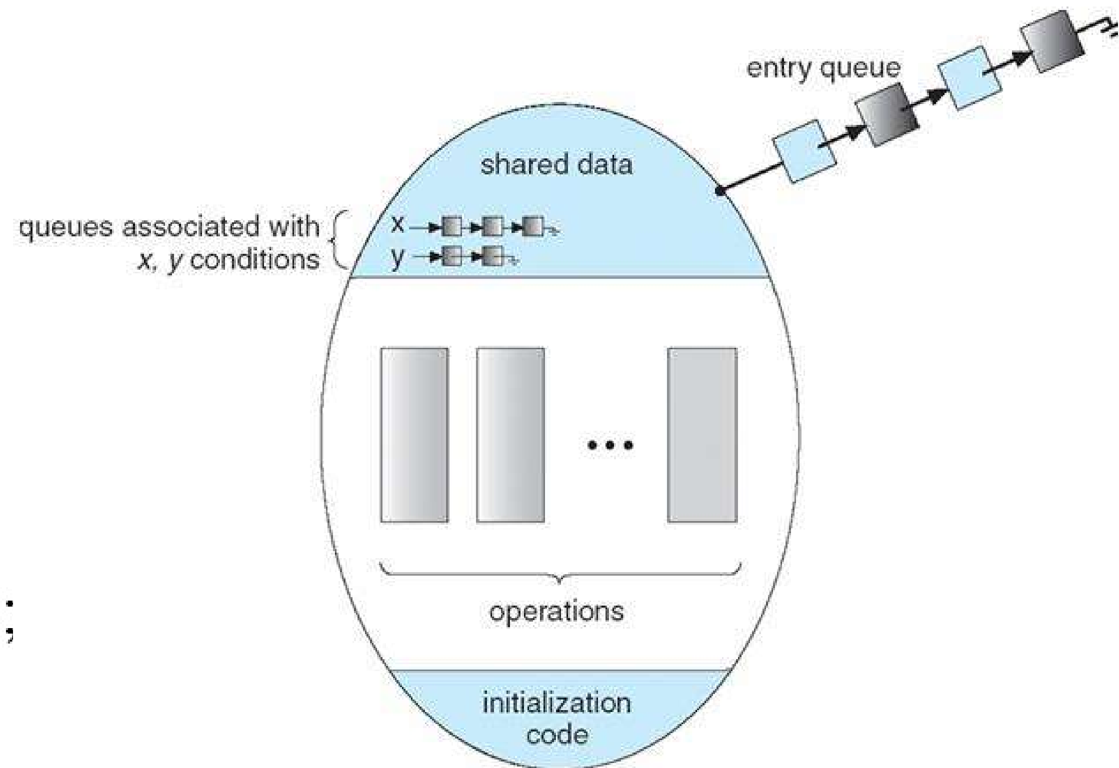
```
monitor MyMonitor {
    // shared variables
    int a;

    // methods
    procedure P_1(...) {
        ...
    }
    procedure P_2(...) {
        ...
    }

    // initialization
    initialize() {
        a = ...
        ...
    }
}
```

Monitor Example

monitor *example*
    integer *i*;
    condition *c*;

    procedure *producer*( );
    .
    .
    .
    end;

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



Producer-Consumer with Monitors

Only one monitor procedure at a time is active. The buffer has *N* slots.

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count – 1;
                if count = N – 1 then signal(full)
        end;

        count := 0;
end 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;
```

POSIX - Pthread Mutex Calls

Some of the Pthreads' calls relating to mutexes.

Thread call	Description
pthread_mutex_init	Create a mutex
pthread_mutex_destroy	Destroy an existing mutex
pthread_mutex_lock	Acquire a lock or block
pthread_mutex_trylock	Acquire a lock or fail
pthread_mutex_unlock	Release a lock

Condition Variables

Thread call	Description
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread_cond_wait	Block waiting for a signal
pthread_cond_signal	Signal another thread and wake it up
pthread_cond_broadcast	Signal multiple threads and wake all of them

Producer-Consumer with Threads

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                      /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                /* used for signaling */
int buffer = 0;                             /* buffer used between producer and consumer */

void *producer(void *ptr)                   /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                        /* put item in buffer */
                pthread_cond_signal(&condc);       /* wake up consumer */
                pthread_mutex_unlock(&the_mutex); /* release access to buffer */
        }
        pthread_exit(0);
}

void *consumer(void *ptr)                   /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                        /* take item from buffer (not shown) and reinitialize */
                pthread_cond_signal(&condp);       /* wake up producer */
                pthread_mutex_unlock(&the_mutex); /* release access to buffer */
        }
        pthread_exit(0);
}


int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Producer-Consumer with N Messages

```
#define N 100                                  /* number of slots in the buffer */

void producer(void)
{
        int item;
        message m;                             /* message buffer */

        while (TRUE) {
                item = produce_item( );        /* generate something to put in buffer */
                receive(consumer, &m);         /* wait for an empty to arrive */
                build_message(&m, item);       /* construct a message to send */
                send(consumer, &m);            /* send item to consumer */
        }
}


void consumer(void)
{
        int item, i;
        message m;

        for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
        while (TRUE) {
                receive(producer, &m);         /* get message containing item */
                item = extract_item(&m);       /* extract item from message */
                send(producer, &m);            /* send back empty reply */
                consume_item(item);            /* do something with the item */
        }
}
```

Readers-Writers: Shared Data Access

Used for structures that do not change frequently

- Readers: Data data does not change
- Writers: Data is changing

Rules

- Multiple Readers allowed simultaneously
- Writers must have exclusive access

Priorities

- Writers or Readers
- Starvation is possible in both cases

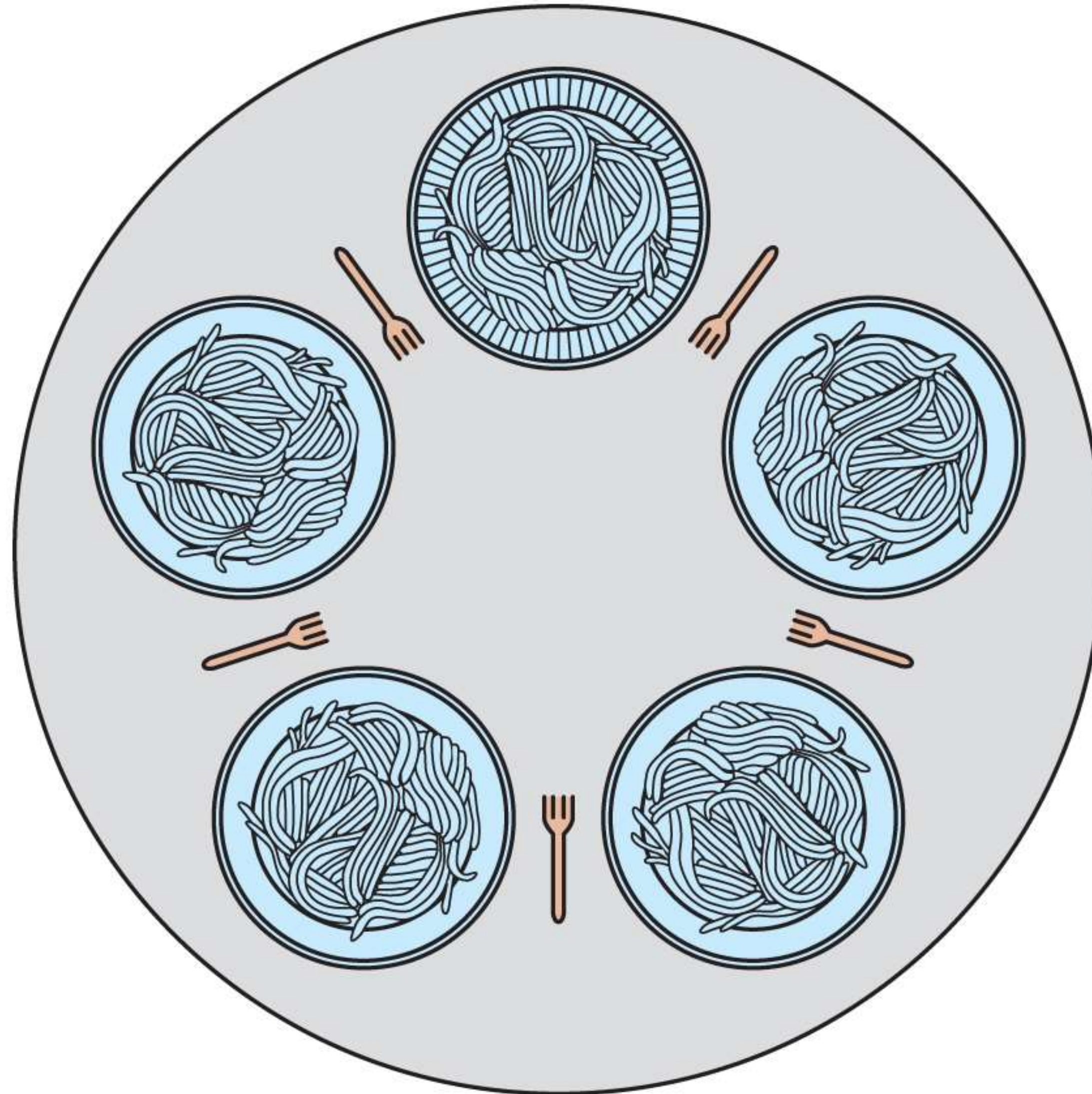
Readers-Writers (readers priority)

```
sema_t mutex = semaphore(1);
sema_t write = semaphore(1);
int readcount = 0;
```

```
read_lock:
    wait(mutex);
    if (++readcount == 1)
        wait(write);
    signal(mutex);

read_unlock:
    wait(mutex);
    if (--readcount == 0)
        signal(write);
    signal(mutex);
```

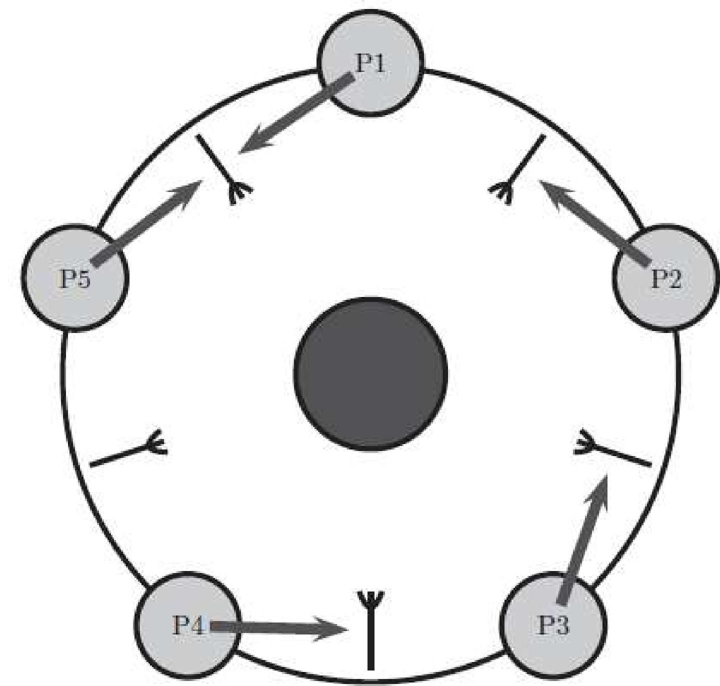
```
write_lock:
    wait(write);

write_unlock:
    signal(write);
```

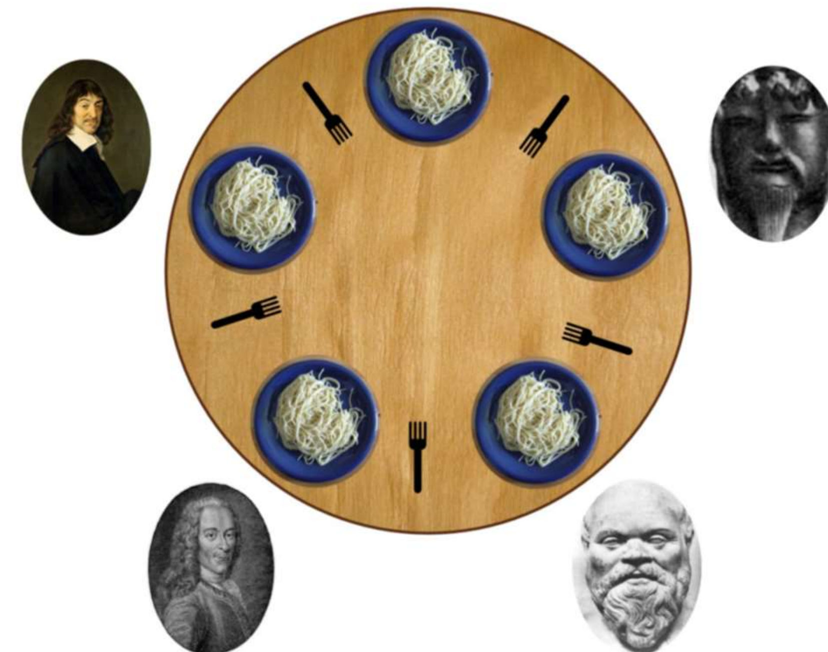
Lunch in the Philosophy Department



Dining Philosophers

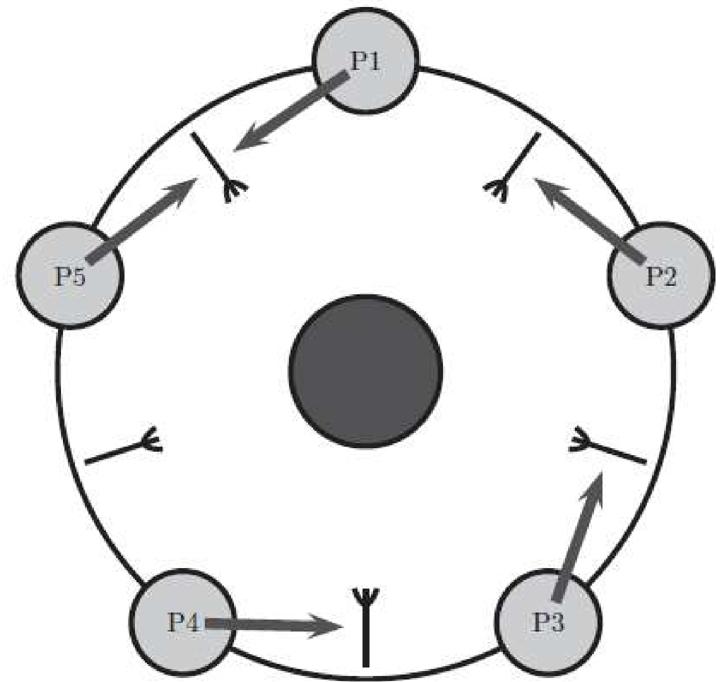
- Dijkstra's resource management problem
- 5 philosophers (eat and think)
- 5 chopsticks
- A hungry philosopher needs 2 chopsticks to eat
- Two philosophers cannot use the same stick
- Avoid deadlock
- Avoid starvation





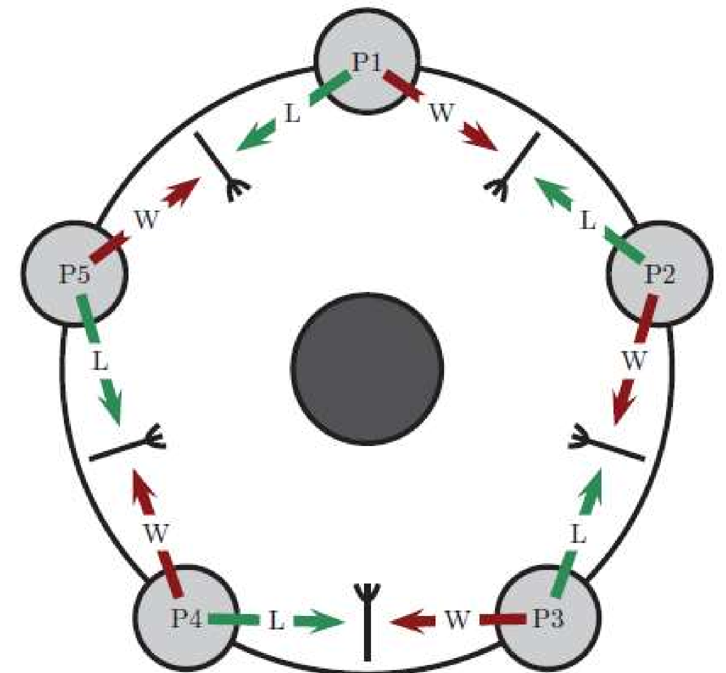
Dining Philosophers

```
#define NEXT(x) ((x+1) % N)
sema_t F[N]; //forks
do {
  think();
  wait(F[i]);
  wait(F[NEXT(i)]);
  eat();
  signal(F[NEXT(i)]);
  signal(F[i]);
} while (TRUE);
```



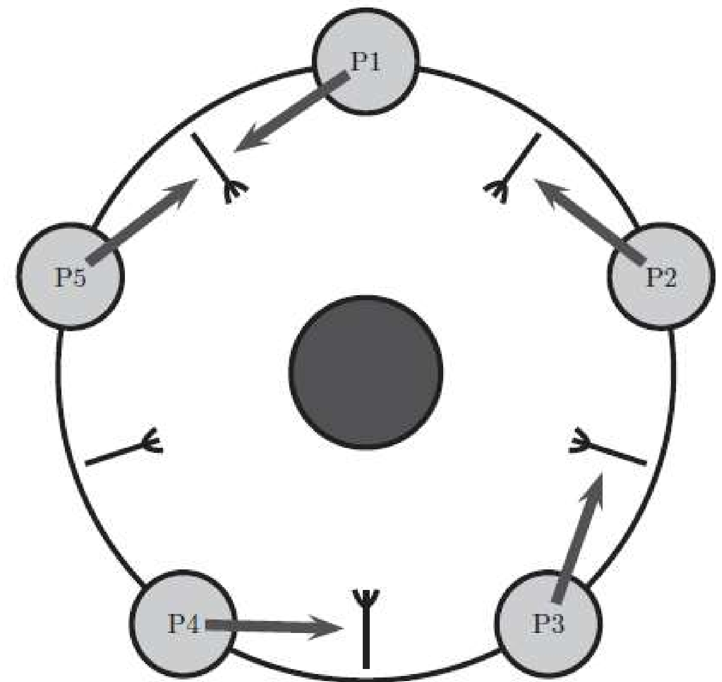
1. P1 παίρνει F1
2. P2 παίρνει F2
3. P3 παίρνει F3
4. P4 παίρνει F4
5. P5 παίρνει F5

⇒ deadlock



Dining Philosophers

- Use a semaphore, initialized to the value 4, so that in any case all 5 philosophers do not claim the sticks
- Note: If one of the philosophers starts to take a stick in a different order than others, then a deadlock cannot be created



Dining Philosophers - Solution

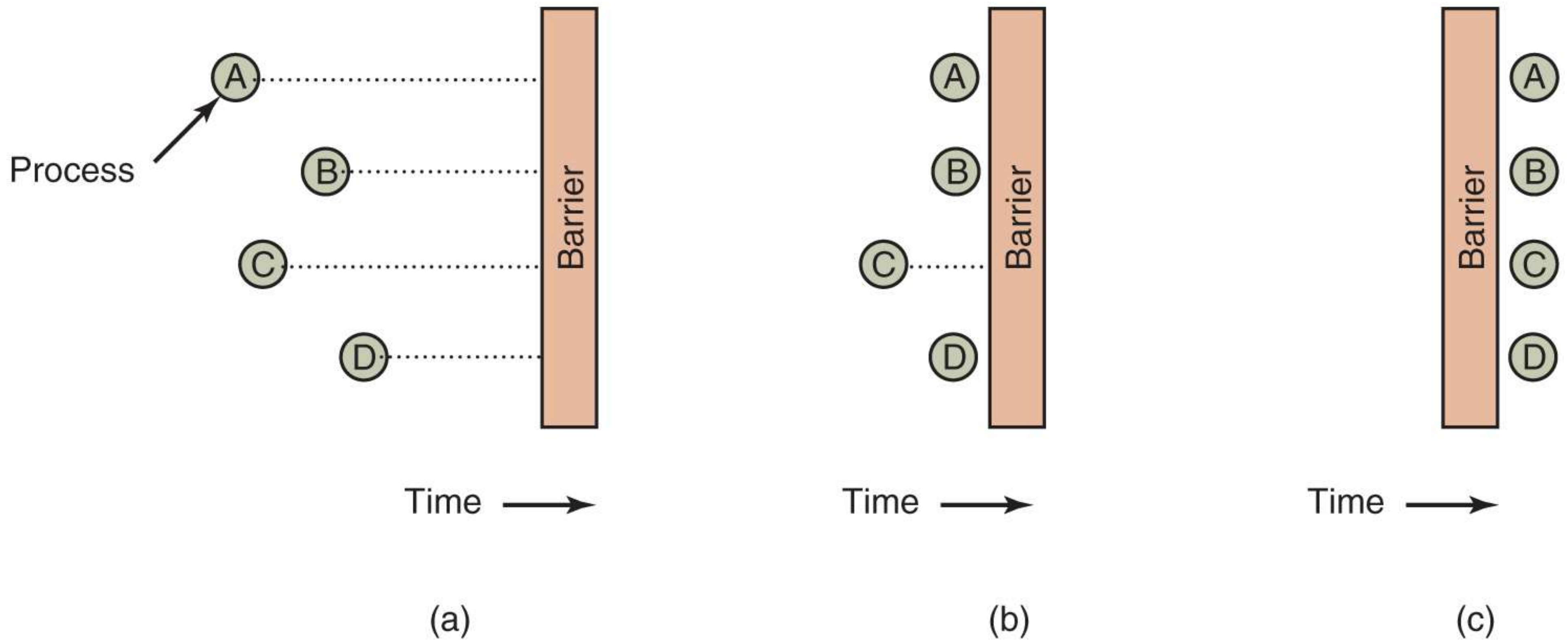
```
monitor DiningPhilosophers {
  enum {T, H, E} state[5];
  condition cond[N];

  void pickup(int i) {
      state[i] = H;
      test(i);
      if (state[i] != E)
          cond[i].wait();
  }

  void putdown(int i) {
      state[i] = T;
      test(PREV(i));
      test(NEXT(i));
  }

  void test(int i){
      if (state[i] == H &&
          state[PREV(i)] != E && state[NEXT(i)] != E){
              state[i] = E;
              cond[i].signal();
      }
  }
} // end of DiningPhilosophers
```

Barrier

(a) Processes approaching a barrier

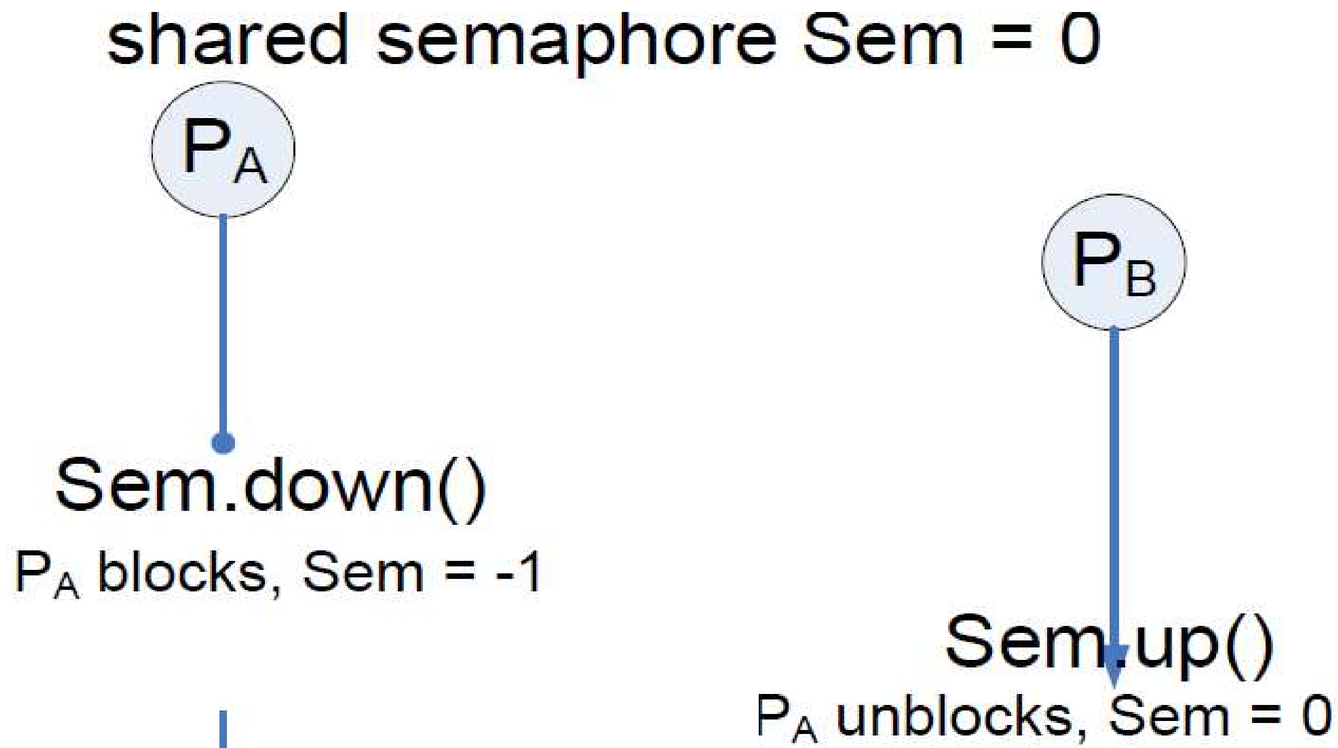
(b) All processes but one blocked at the barrier.

(c) When the last process arrives at the barrier, all of them are let through



Barriers with semaphores

- If we want to block a process until something else occurs, we use a semaphore initialized to 0



Barrier Example

Suppose PA has spawned PB, PC, and PD and we wish PA to wait until its children have terminated