ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Real-Time Systems for Automation M

9. Synchronization

Notice

The course material includes slides downloaded from:

http://codex.cs.yale.edu/avi/os-book/

(slides by Silberschatz, Galvin, and Gagne, associated with Operating System Concepts, 9th Edition, Wiley, 2013)

and

http://retis.sssup.it/~giorgio/rts-MECS.html

(slides by Buttazzo, associated with Hard Real-Time Computing Systems, 3rd Edition, Springer, 2011)

which have been edited to suit the needs of this course.

The slides are authorized for personal use only.

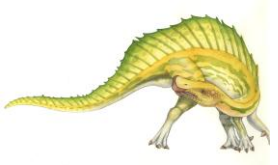
Any other use, redistribution, and any for profit sale of the slides (in any form) requires the consent of the copyright owners.

Chapter 6: Process Synchronization

- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization

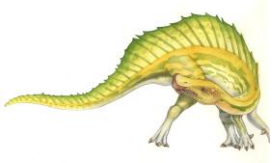
Chapter 7: Deadlocks

- Deadlock Characterization
- Methods for Handling Deadlocks: Prevention, Avoidance, Detection and Recovery



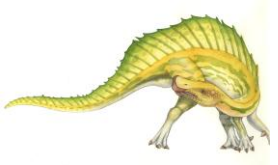
Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems
- To provide a methodology for achieving a correct synchronization among tasks using such tools



Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
    - Voluntary interruption or pre-emption
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes



Bounded Buffer Reloaded

- Recall:
  - Shared data
  - Producer
  - Consumer

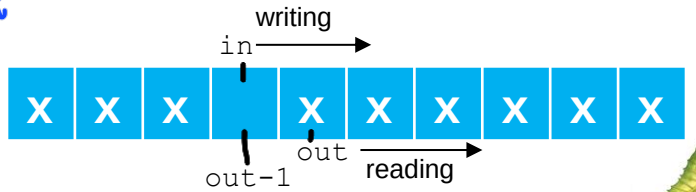
```
#define SIZE 10
typedef struct { … } item;
item buffer[SIZE];
// pointers to next element to write/read:
int in = 0, out = 0;
```

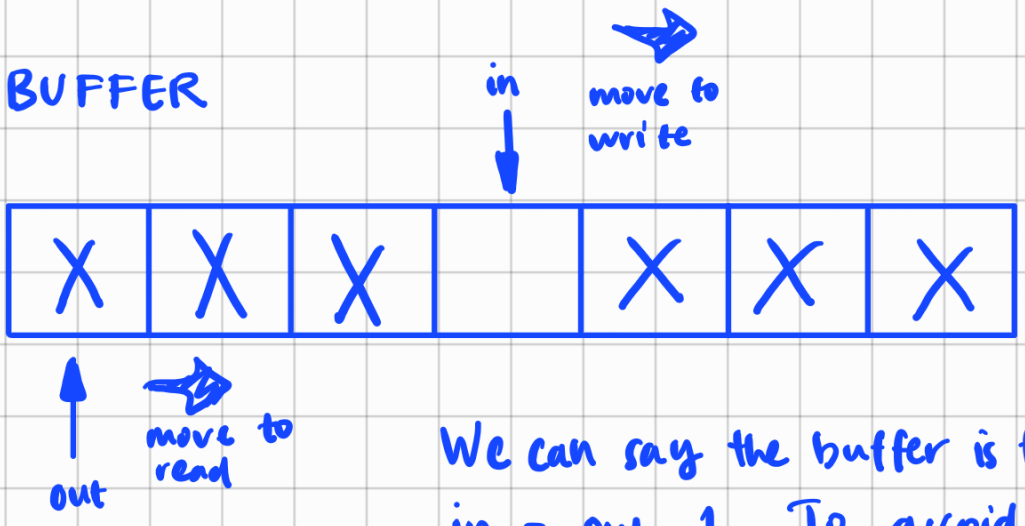
```
/* producer */
item next_produced;
while (1) { // while true
  /* produce an item … */
  // while the buffer is full
  while (((in + 1) % SIZE) == out)
    ; /* do nothing */
  // when is not full, write
  buffer[in] = next_produced;
  in = (in + 1) % SIZE;
}
```

```
/* consumer */
item next_consumed;
while (1) {
  // while buffer empty
  while (in == out)
    ; /* do nothing */
  // when is not empty, read
  next_consumed = buffer[out];
  out = (out + 1) % SIZE;
  /* … consume the item */
}
```

Without counter, one of them only modifying the input

- Solution is correct
- can only use **`SIZE-1`** elements
  - `out-1` position never overwritten

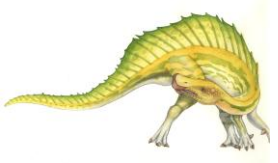




We can say the buffer is full when in = out - 1. To avoid full, we can deploy COUNTER and associate it to the buffer SIZE.

Bounded Buffer Reloaded

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer **`counter`** that keeps track of the number of full buffers.
  - Initially, **`counter`** is set to 0.
  - Then, **`counter`** is incremented by the producer after it produces (in) a new buffer and it is decremented by the consumer after it consumes (the content of) a buffer.



Bounded Buffer with Counter

```
/* producer */
item next_produced;
while (1) { // while true
  /* produce an item … */
  while (counter == SIZE)
    ; /* do nothing */
  buffer[in] = next_produced;
  in = (in + 1) % SIZE;
  counter++;
}
```

THE BUFFER IS FULL!

```
/* consumer */
item next_consumed;
while (1) {
  while (counter == 0)
    ; /* do nothing */
  next_consumed = buffer[out];
  out = (out + 1) % SIZE;
  counter--;
  /* … consume the item */
}
```

THE BUFFER IS EMPTY!

- Both producer and consumer manipulate the same shared variable
  - Load from memory to CPU, change, store in the memory
- Possible conflicts!

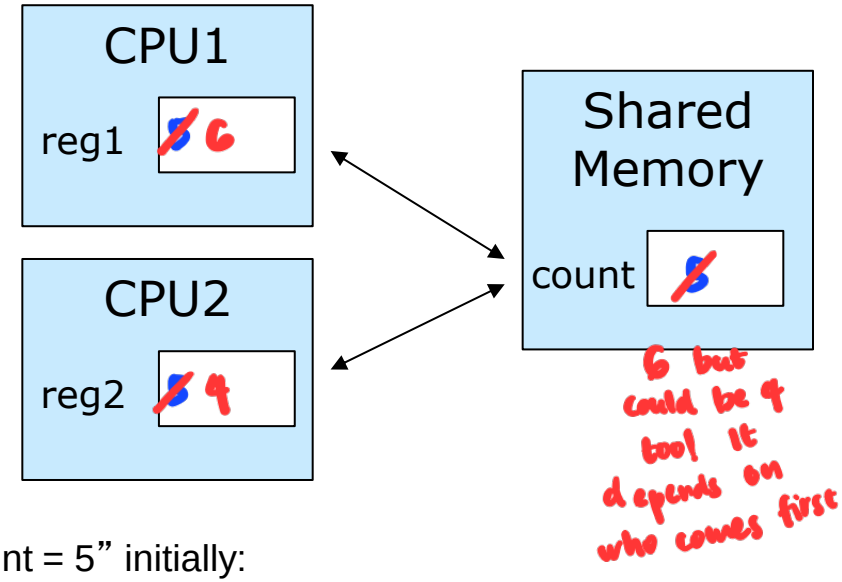
Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

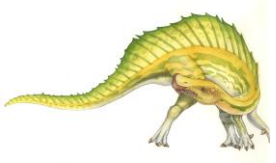


- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in **its** critical section, no other may be in **their** critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**



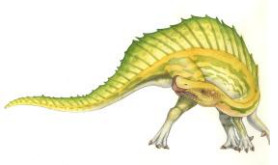
Critical Section

- General structure of process p_i is:

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

check first the I/O or files

shared var. and resource



Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

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

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed
- No assumption concerning **relative speed** of the ***n*** processes

- kernel code is subject to several possible race conditions (access to shared data structures). Two approaches depending on if kernel is preemptive or non-preemptive
  - **Preemptive** – allows preemption of process when running in kernel mode
  - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode
    - Risk process will run for an excessively long time

solutions → preemp. / non-preemp.

Peterson's Solution

- Good algorithmic description of solving the problem

  only work on two processes

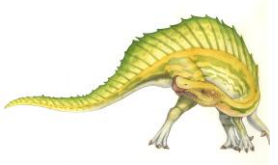
- Two-process solution

- Assume that the **`load`** and **`store`** instructions are **atomic**;
  - Atomic instructions: they cannot be interrupted

- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`** only two literally!

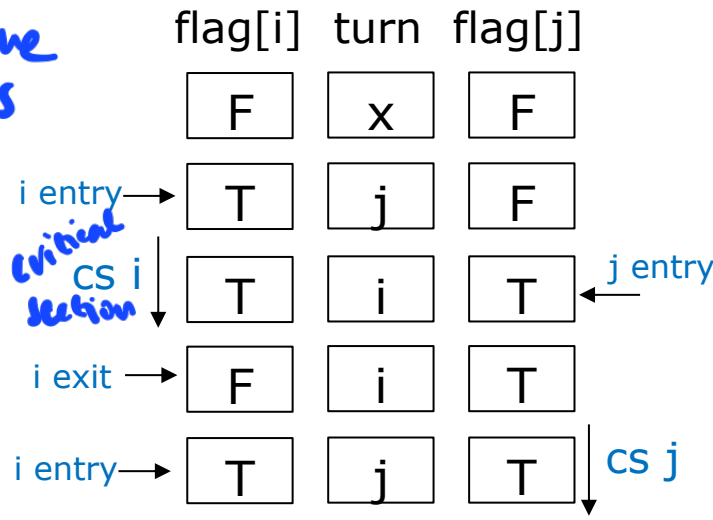
- The variable **`turn`** indicates whose turn it is to enter the critical section

- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = TRUE`** implies that process P_i is ready!



Algorithm for Process P_i

```
FOREVER {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    critical section
    flag[i] = FALSE;
    remainder section
};
```

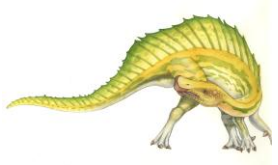


- **Mutual exclusion** is preserved (turn is either j or i) ✔
- **Progress** requirement is satisfied (a process waits only if the other is in the CS)
- **Bounded-waiting** requirement is met (once a process exits the CS it cannot reenter a second time if the other has already requested access)
- needs to be extended to deal with more than **2 processes**
- it is **only a software solution**. Modern architectures may reorder some instructions. Need hardware support to explicitly forbid this.

WE NEED HARDWAR SYNC FOR >2 PROCESSES

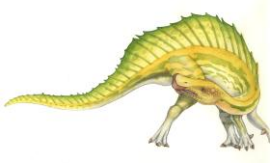
Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking** → if it only has 1 CPU
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words



Solution to Critical-section Problem Using Locks

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```



test_and_set() Instruction

THIS WILL ABSOLUTELY GUARANTEE ONLY ONE PROCESS HAS THE LOCK.

- Check the lock and close it as a single operation
- Hardware instruction, executed atomically

- Definition:

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = TRUE;
        return rv:
}
```

evaluate the var. value and set it

this is the lock

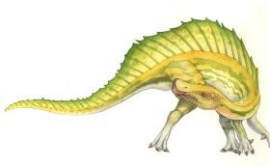
value before TRUE

(TEST) if nobody has the lock

set the target = TRUE (I have the lock!)

CS

- Target=lock; true=locked
- If the lock was open, close it and return open.
- If the lock was closed, I keep it closed and return closed.



Solution (?) using `test_and_set()`

- Shared Boolean variable `lock`, initialized to **`FALSE`**

```
do {
   while (test_and_set(&lock))
        ; /* do nothing */
        /* critical section */
   lock = FALSE;
        /* remainder section */
} while (TRUE);
```

check if someone has the lock

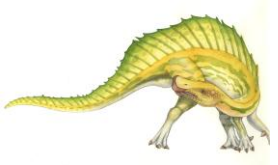
once no one has the lock, we have the lock, then do the critical section

```
boolean test_and_set
   (boolean *target){
   boolean rv = *target;
   *target = TRUE;
   return rv:
   }
```

- **Mutual exclusion** is preserved
- **Progress** requirement is satisfied
- Works even with **more than 2 processes**
- it is **a hardware solution.** Executed atomically by the CPU: no race condition
- **Bounded-waiting** requirement not met!

LOCK OPEN if LOCK = FALSE
LOCK CLOSE if LOCK = TRUE

We can't make sure a process will have the lock



Bounded-waiting Mutual Exclusion

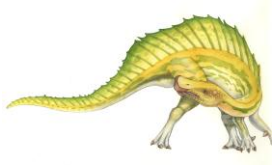
- Shared Boolean variables `lock` and `flag[n]` initialized to **FALSE** (flag[n]: array of flags)
- No release of the lock if another process is waiting (instead of leaving the lock open for everyone, we give the lock to the next process in the queue)

```
do {
    flag[i] = TRUE; /* waiting to be granted access to critical section */
    while (flag[i] && test_and_set(&lock))
          ; /* do nothing */
    flag[i] = FALSE;
            /* critical section */
    /* check if there is a flag */
    j = (i + 1) % n;
    while ((j != i) && !flag[j])
       j = (j + 1) % n;
    if (j == i) /* round completed, no flags: open lock */
       lock = FALSE; (release the lock for everyone)
    else /* one flag found: just change that, lock remains closed */
       flag[j] = FALSE; (give the lock to the next process in the queue)
            /* remainder section */
} while (TRUE);
```

Mutex Locks

allow a process to have a lock using an API

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build **software tools** to solve critical section problem
- Simplest is **mutex lock**
- Protect critical regions with it by first **`acquire()`** a lock then **`release()`** it
  - Boolean variable indicating if lock is available or not
  - Typically, the process acquiring the lock is the only one who can release it
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions



acquire() and release()

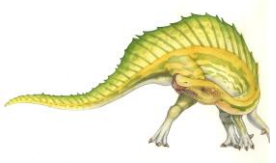
```
acquire() {
    while (!available)
        ; /* busy wait */
    available = FALSE;
}
release() {
    available = TRUE;
}
```

checking whether the key is available. Once it's available, go on!

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

- Mutex can be implemented with or without busy wait
- Mutex with busy wait called **spinlock**
  - Not necessarily a useless solution
    - No context switch required when a process must wait on a lock
    - **Useful when locks expected for short periods of time**
    - Especially in **multiprocessor** systems

remember Round Robin with certain time quantum



POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

initialize default behavior

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

Mutex Usage: Discussion

- Consider the following code fragments.
  Why are mutex locks used? Discuss.

```
#include <pthread.h>

pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
long long count;

void increment_count()
{
  pthread_mutex_lock(&count_mutex);
    count = count + 1;
  pthread_mutex_unlock(&count_mutex);
}

long long get_count()
{
  long long c;
  pthread_mutex_lock(&count_mutex);
    c = count;
  pthread_mutex_unlock(&count_mutex);
  return (c);
}
```

With this method, we can make sure that every process does not use var. count at the same time

acquired lock

released lock

1 2 3

new var c

set c = count

why not return(count)? We don't want to risk a process that we don't have the mutex yet

if we don't introduce mutex, we don't know if increment_count() runs first or last! Also applies to get_count

MUTEX IS BASED ON BINARY VALUE

Semaphore

with semaphore, we can allow more processes at the same time

- Synchronization tool that provides more sophisticated ways (than mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** – integer variable that can be modified only by two indivisible (atomic) operations: **`wait()`** and **`signal()`**
  - Sometimes **`wait`**/**`signal`** are called **`wait`**/**`post`**, **`take`**/**`give`**, or also **`P`**/**`V`**
  - No semaphore "owner" (while Mutex has an owner)
  - can be initialized to 0

Actual semaphore implementation without busy wait

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

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

if the semaphore decreases below zero, it has to wait!

these processes are atomic

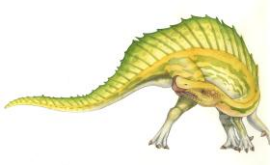
Semaphore Usage

- **Binary semaphore** – integer value can range only between 0 and 1
  - Similar to **mutex lock,** but no "ownership"
  - Can solve various synchronization problems
  - Consider P_1 and P_2 that require S_1 to happen before S_2 (precedence relation)
    - `synch` initialized at 0

```
P1:
    S1;
    signal(synch);

P2:
    wait(synch);
    S2;
```

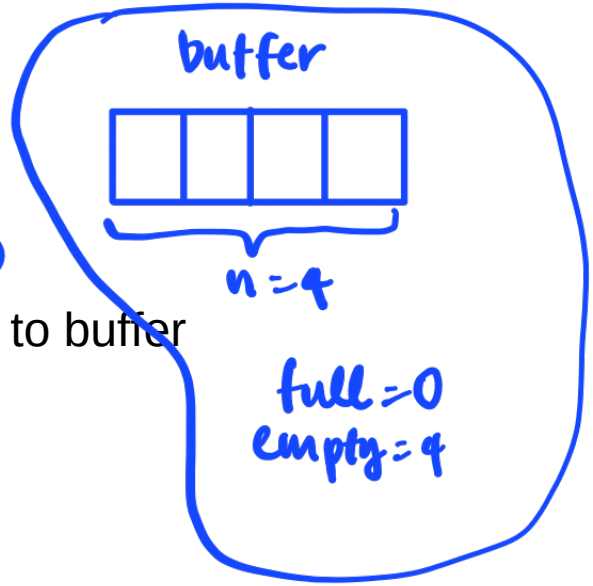
increment the semaphore

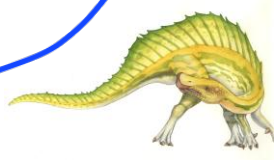
- **Counting semaphore** – integer value can range over an unrestricted domain



Bounded-Buffer Problem

- Buffer with ***n*** slots, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
  - Used to grant mutually exclusive access to buffer
- Semaphore **`full`** initialized to the value 0
  - Number of full slots
- Semaphore **`empty`** initialized to the value n
  - Number of empty slots





Bounded-Buffer Problem

- The structure of the producer process

```
do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next_produced to the buffer */
    ...
    signal(mutex);
    signal(full);
} while (TRUE);
```

wait(empty); → if there is at least one empty slot, decrease the value

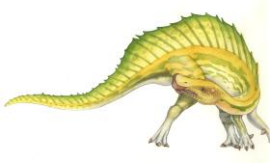
wait(mutex); → If not, wait until acquired the mutex

acquire

release

signal(mutex); → release the mutex

signal(full); → increment the semaphore



Bounded-Buffer Problem

- The structure of the consumer process

```
do {
    wait(full);
    wait(mutex);
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next_consumed */
        ...
} while (TRUE);
```

if we execute this first than wait(full): the consumer waits / reads an empty buffer

POSIX Semaphores (unnamed)

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

0: semaphore shared between the threads of a process. `sem` needs to be a global variable
1: semaphore shared between processes. Needs shared memory

Global var

initialize the semaphore beginning

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

- **`sem_destroy()`** to destroy the semaphore

POSIX Semaphores (named)

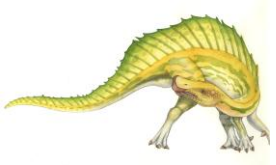
- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

semaphore name = "SEM"

- Another process can access the semaphore by referring to its name **SEM**.
- Acquiring and releasing the semaphore:

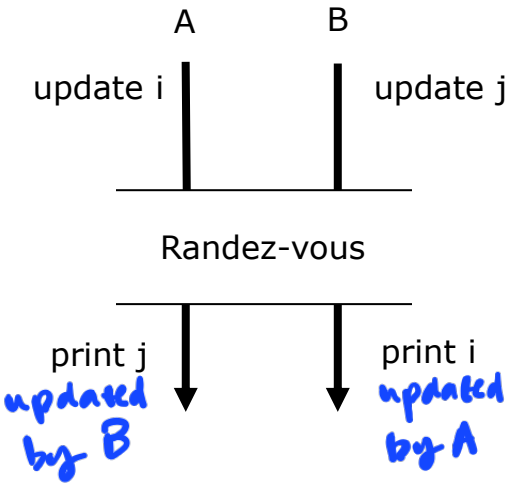
```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```



Semaphore Exercise: Rendez-vous

- Consider two concurrent POSIX threads executing two functions, **`runnerA`** and **`runnerB`**, working on two shared variables, **`i`** and **`j`**:

```
void *runnerA(void *param) {
    i = 5; // i updated
    printf("j: %d\n", j);
    pthread_exit(NULL);
    }
void *runnerB(void *param) {
    j = 7; // j updated
    printf("i: %d\n", i);
    pthread_exit(NULL);
    }
```



- How can semaphores be used in order to **ensure** that the two threads display the **updated** value of **`i`** and **`j`**?

Rendez-vous: Discussion

- Consider the following solutions. How do they differ? Discuss.

```
// i_done and j_done are two semaphores
// assume i_done and j_done been both initially taken (not available)
```

initially = 0

1)

```
void *runnerB(void *param) {
  j = 7; // j updated
  sem_post(&j_done);
  sem_wait(&i_done);
  printf("i: %d\n", i);
  pthread_exit(NULL);
}
```

post j is done
wait for i is done

```
void *runnerA(void *param) {
  i = 5; // i updated
  sem_post(&i_done);
  sem_wait(&j_done);
  printf("j: %d\n", j);
  pthread_exit(NULL);
}
```

post i is done
wait for j is done

2)

```
void *runnerB(void *param) {
  j = 7; // j updated
  sem_wait(&i_done);
  sem_post(&j_done);
  printf("i: %d\n", i);
  pthread_exit(NULL);
}
```

works but asymetric

```
void *runnerA(void *param) {
  i = 5; // i updated
  sem_post(&i_done);
  sem_wait(&j_done);
  printf("j: %d\n", j);
  pthread_exit(NULL);
}
```

3)

```
void *runnerB(void *param) {
  j = 7; // j updated
  sem_wait(&i_done);
  sem_post(&j_done);
  printf("i: %d\n", i);
  pthread_exit(NULL);
}
```

wait forever

```
void *runnerA(void *param) {
  i = 5; // i updated
  sem_wait(&j_done);
  sem_post(&i_done);
  printf("j: %d\n", j);
  pthread_exit(NULL);
}
```

Rendez-vous: Discussion

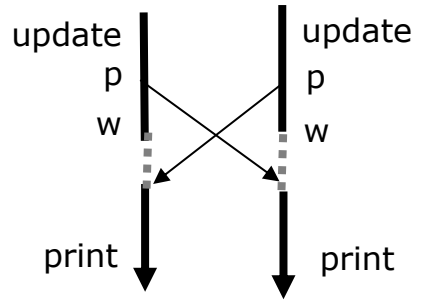
- Consider the following solutions. How do they differ? Discuss.

```
// i_done and j_done are two semaphores
// assume i_done and j_done been both initially taken (not available)
```

1)

```
void *runnerB(void *param) {
  j = 7; // j updated
  sem_post(&j_done);
  sem_wait(&i_done);
  printf("i: %d\n", i);
  pthread_exit(NULL);
}
```

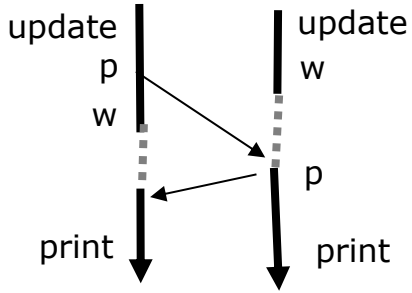
```
void *runnerA(void *param) {
  i = 5; // i updated
  sem_post(&i_done);
  sem_wait(&j_done);
  printf("j: %d\n", j);
  pthread_exit(NULL);
}
```



2)

```
void *runnerB(void *param) {
  j = 7; // j updated
  sem_wait(&i_done);
  sem_post(&j_done);
  printf("i: %d\n", i);
  pthread_exit(NULL);
}
```

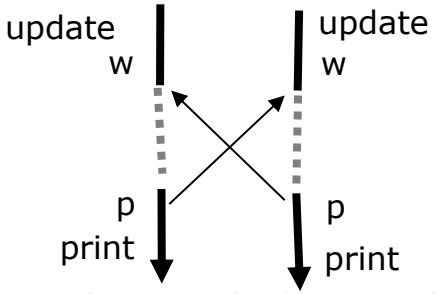
```
void *runnerA(void *param) {
  i = 5; // i updated
  sem_post(&i_done);
  sem_wait(&j_done);
  printf("j: %d\n", j);
  pthread_exit(NULL);
}
```



3)

```
void *runnerB(void *param) {
  j = 7; // j updated
  sem_wait(&i_done);
  sem_post(&j_done);
  printf("i: %d\n", i);
  pthread_exit(NULL);
}
```

```
void *runnerA(void *param) {
  i = 5; // i updated
  sem_wait(&j_done);
  sem_post(&i_done);
  printf("j: %d\n", j);
  pthread_exit(NULL);
}
```

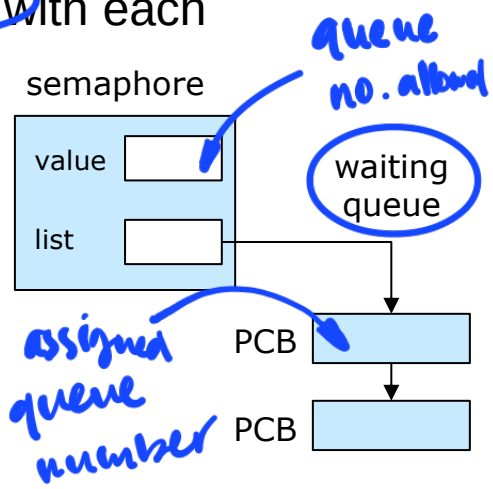


Semaphore Implementation with no Busy waiting

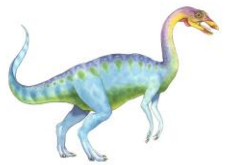
```
wait(S.value){
     while (S.value <= 0);
     S.value--;
}
```

Applications may spend lots of time in critical sections. In general, busy wait is **not a good solution**

- To avoid busy wait: need to associate a **waiting queue** with each semaphore
- Each semaphore has two data items:
  - value (of type integer)
  - pointer to next record in the list (of PCB)



- Two operations:
  - `block()` – place the process invoking the operation on the appropriate waiting queue
  - `wakeup()` – remove one of processes in the waiting queue and place it in the ready queue



Semaphore Implementation with no Busy waiting

structural object "so-called"

```
typedef struct{
    int value;
    struct process *list;
} semaphore;
```

Not possible in `semaphore.h`

- Now `s.value` can be negative
- if so, `abs(s.value)` would tell you how many processes are in the queue

```
wait(semaphore *S) {
    S->value--;

    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

decrement

added to list

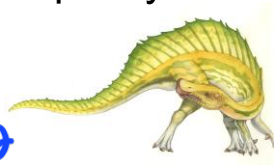
block the process

```
signal(semaphore *S) {
    S->value++;

    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

state the process

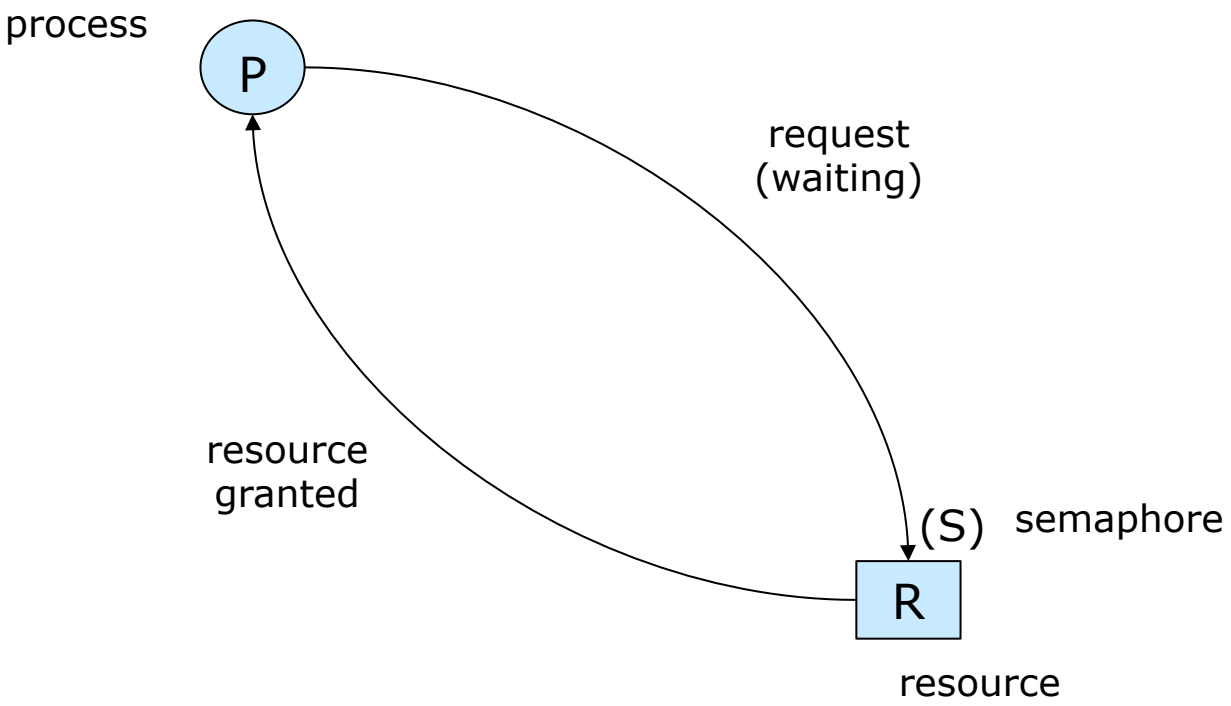
- `block()` blocks the process who calls it
- `wakeup()` awakes someone else: you don't know who. However, you can specify the policy to manage the queue
- …how to implement them?

we can know from the semaphore how many processes are waiting



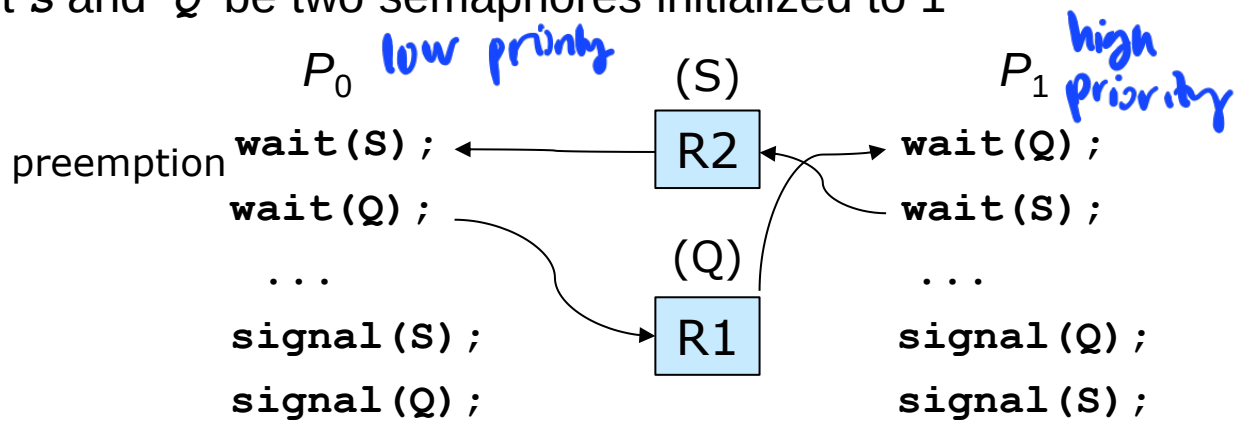
Graphical notation

- Resource request/allocation graph

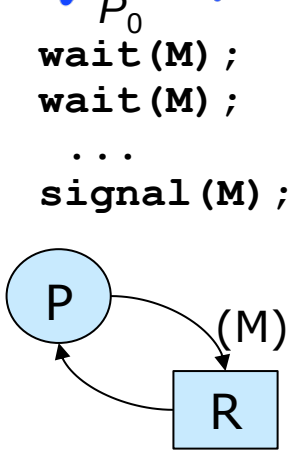


Deadlock and Starvation

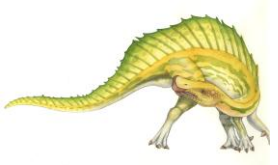
- **Deadlock** – (one) two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let *s* and *q* be two semaphores initialized to 1



deadlock by itself

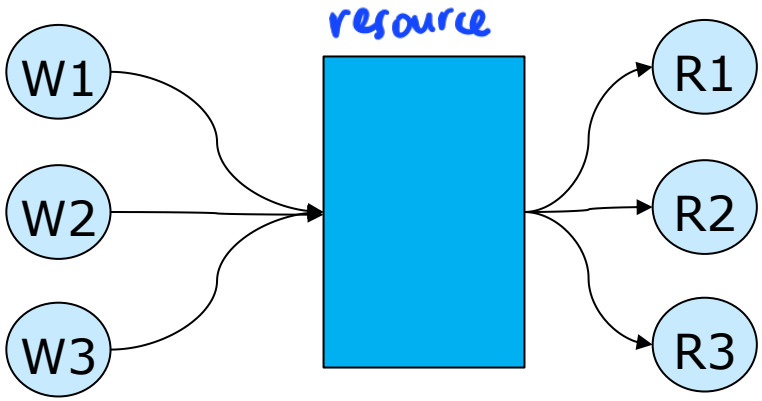


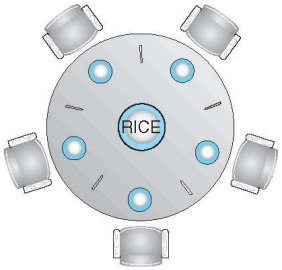
- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

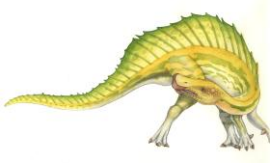


Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem







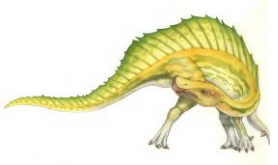
Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem

- The structure of a writer process

```
do {
   wait(rw_mutex);
      ...
   /* writing is performed */
      ...
   signal(rw_mutex);
} while (TRUE);
```



Readers-Writers Problem

- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1) // this is the only reader
                wait(rw_mutex); // blocks the writers
        signal(mutex);

                ...
        /* reading is performed */

                ...

        wait(mutex);
        read_count--;
        if (read_count == 0) // this is the last reader
                signal(rw_mutex); // unlocks the writers
        signal(mutex);
} while (TRUE);
```

acquired mutex only for reader

increment the reader

Critical section with reader

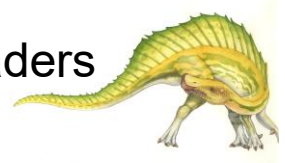
release the mutex for another reader

Writer blocked

Critical section with writer

Critical section with reader

…this solution favors the readers

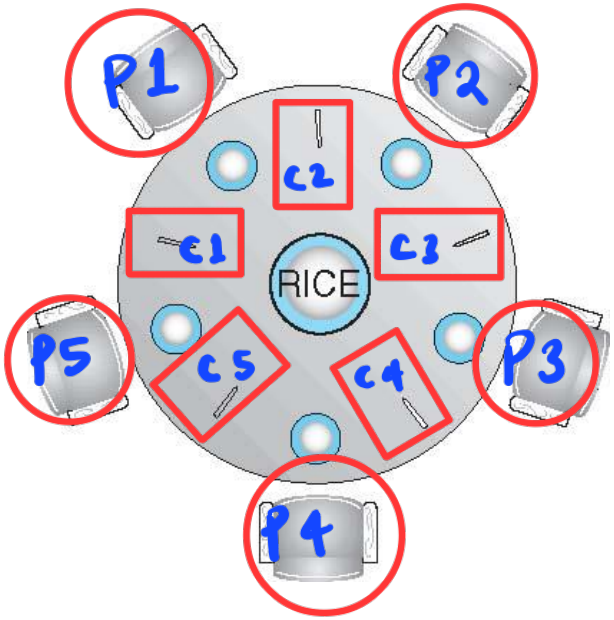


Readers-Writers Problem Variations

- *First* version – no reader kept waiting unless writer has permission to use shared object → writer may starve
- *Second* version – once writer is ready, it performs write asap → readers may starve
- Both may have starvation leading to even more variations
- Many kernels provide **readers/writer locks** (no need to implement). E.g., POSIX defines:
  - `pthread_rwlock_t`
  - `pthread_rwlock_rdlock, pthread_rwlock_wrlock`
  - `pthread_rwlock_unlock`

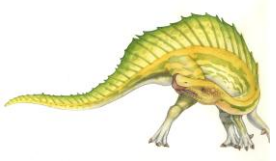
Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - 5 chopstick (pairwise shared resources)



P1
take (C1)
take (C2)
eat
release (C1)?
release (C2)

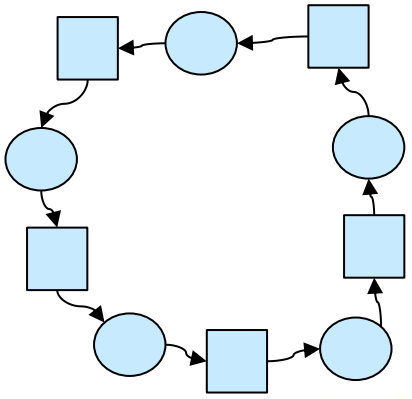
a phil. needs two chopsticks to eat.



Dining-Philosophers Problem Algorithm

- A semaphore for each shared resource (i.e., chopstick)
- The structure of Philosopher *i*:

```
do  {
                    //  think

            wait ( chopstick[i] );
            wait ( chopstick[ (i + 1) % 5] );

                    //  eat

            signal ( chopstick[i] );
            signal (chopstick[ (i + 1) % 5] );
} while (TRUE);
```



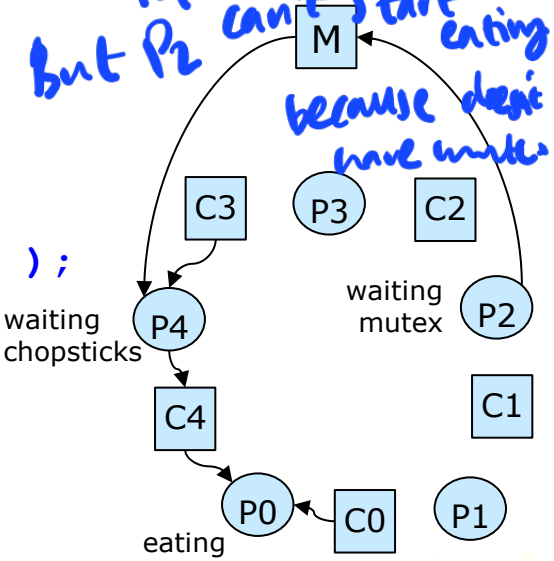
- What is the problem with this algorithm?
- What are possible solutions?

Dining-Philosophers – Solution(?) 1

- Protecting the chopstick request with a critical section

```
do  {
                    //  think
            wait (m)
            wait ( chopstick[i] );
            wait ( chopstick[ (i + 1) % 5] );
            signal (m)
                    //  eat

            signal ( chopstick[i] );
            signal (chopstick[ (i + 1) % 5] );
} while (TRUE);
```

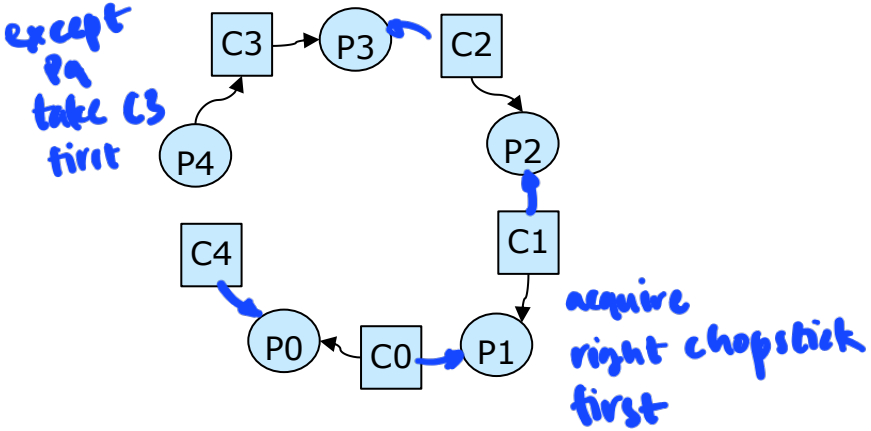
P_4 has the mutex so once P_0 is finished P_4 starts eating. But P_2 can't start eating because deadlock



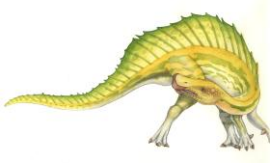
- Solves the deadlock
- **may prevent concurrency** only one philosopher can take two chopsticks at one time

Dining-Philosophers – Solution(?) 2

- Reverse the order of chopstick pickup for some philosophers
  - Some philosopher picks left first, some picks right first



- Solves the deadlock
- Allows concurrency
- Is it a “fair” strategy?



Tanenbaum's Solution With Semaphores

```
enum {THINKING, EATING, HUNGRY} state[5];
semaphore self[5]; // initially: 0
semaphore mutex; // initially: 1

/* i-th philosopher */
while(TRUE) {
    /* THINK */
    pick_up(i);
    /* EAT */
    put_down(i);
}
```

```
pick_up(i) {
    wait(mutex);
    state[i] = HUNGRY; // I want to eat
    test(i); // check if I can eat
    signal(mutex);
    wait(self[i]); // wait permission to eat
}
```

```
int left(i) { return (i+4)%5; }
int right(i) { return (i+1)%5; }

put_down(i) {
    wait(mutex);
    state[i] = THINKING;
    test(right(i)); // try give permission right
    test(left(i)); // try give permission left
    signal(mutex);
}
```

```
test(i) {
    if(state[i] == HUNGRY
       && state[left(i)] != EATING
       && state[right(i)] != EATING) {
        state[i] = EATING;
        signal(self[i]); // give permission
    }
}
```

Tanenbaum's Solution With Semaphores

- Solves the **deadlock:** now philosophers no longer wait on 2 semaphores (one for each chopstick).
  - It is no longer possible that a philosopher holds one resource and waits for the other.
  - philosophers actually wait on two semaphores, mutex and self, but the wait on self is always outside the mutex's critical section.
- Allows **concurrency:** philosophers on opposite sides of the table can eat concurrently
- Is **fair:** all philosophers act the same way. No unbalance.

The philosopher's state recalls the **`flag[i]`** of Peterson's solution

In Peterson: two states, flag true from entry to exit

Here three states, entry, cs, the rest

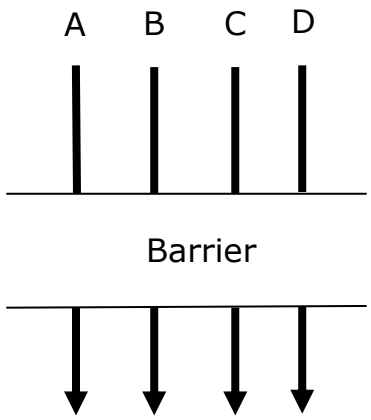
Semaphore Exercise: Barrier

- Extend the rendez-vous to more than two threads.

```
// // i_done and j_done are two semaphores
// assume i_done and j_done been both initially taken (not available)
```

```
void *runnerB(void *param) {
  j = 7; // j updated
  sem_post(&j_done);
  sem_wait(&i_done);
  printf("i: %d\n", i);
  pthread_exit(NULL);
}
```

```
void *runnerA(void *param) {
  i = 5; // i updated
  sem_post(&i_done);
  sem_wait(&j_done);
  printf("j: %d\n", j);
  pthread_exit(NULL);
}
```



- Many kernels provide **barrier** (no need to implement). In POSIX:
  - `pthread_barrier_t b;`
  - `pthread_barrier_init(&b,NULL,4);`
  - `pthread_barrier_wait(&b);`

Possible solution:

1) Sem 1 : mutex
2) Sem 2 : continue or not

Counter → how many processes are waiting

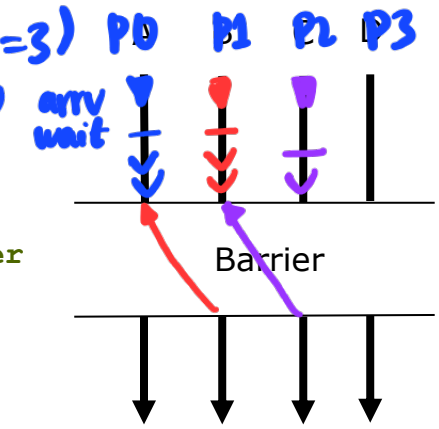
↓

If the number meets the value we want, wake up a process

Semaphore Exercise: Barrier

- Extend the rendez-vous to more than two threads.
  - Use of two semaphores: mutex (init=1) and bar (init=0)
  - Use of a counter initialized to 0 to represent processes at the barrier

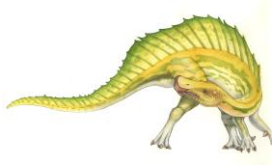
```
void *runner(void *param) {
  /* before barrier */

  sem_wait(&m);
  counter++; // increment counter
  sem_post(&m);

  if (counter==N) // if all process are at the barrier
     sem_post(&bar); // wake up one
  sem_wait(&bar); // wait at the barrier

  sem_post(&bar); // wake up another process

 /* after barrier */
}
```



- Many kernels provide **barrier** (no need to implement). In POSIX:
  - `pthread_barrier_t b;`
  - `pthread_barrier_init(&b,NULL,4);`
  - `pthread_barrier_wait(&b);`

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation
  - wait (mutex)… wait(semaphore)… signal(mutex)
  - …
- Waiting for conditions inside critical section is problematic!

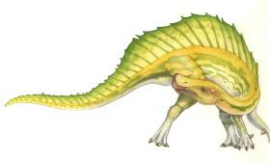


Quizzes

- A wrong wait/signal sequence causes starvation — always — False, sometimes it's fine
- A wrong wait/signal sequence may cause a deadlock — True
- A deadlock-free solution is guaranteed to not cause starvation — False
- Spinlocks are effective especially on single-processor computer systems — False, it works on Multi Cores
- Critical sections should contain as many instructions as possible — False, should be atomic
- Peterson's solution to the critical section problem meets the bounded waiting requirement — True, it allows another process to go first

Understanding Deadlock: Model

- System consists of resources
- Resource **types** $R_1, R_2, \ldots, R_m$
  - *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
  - e.g. a disk partitioned into 3 is modeled as 3 instances of resource type disk
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**



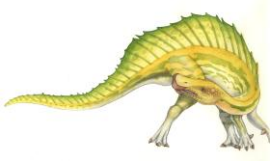
Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

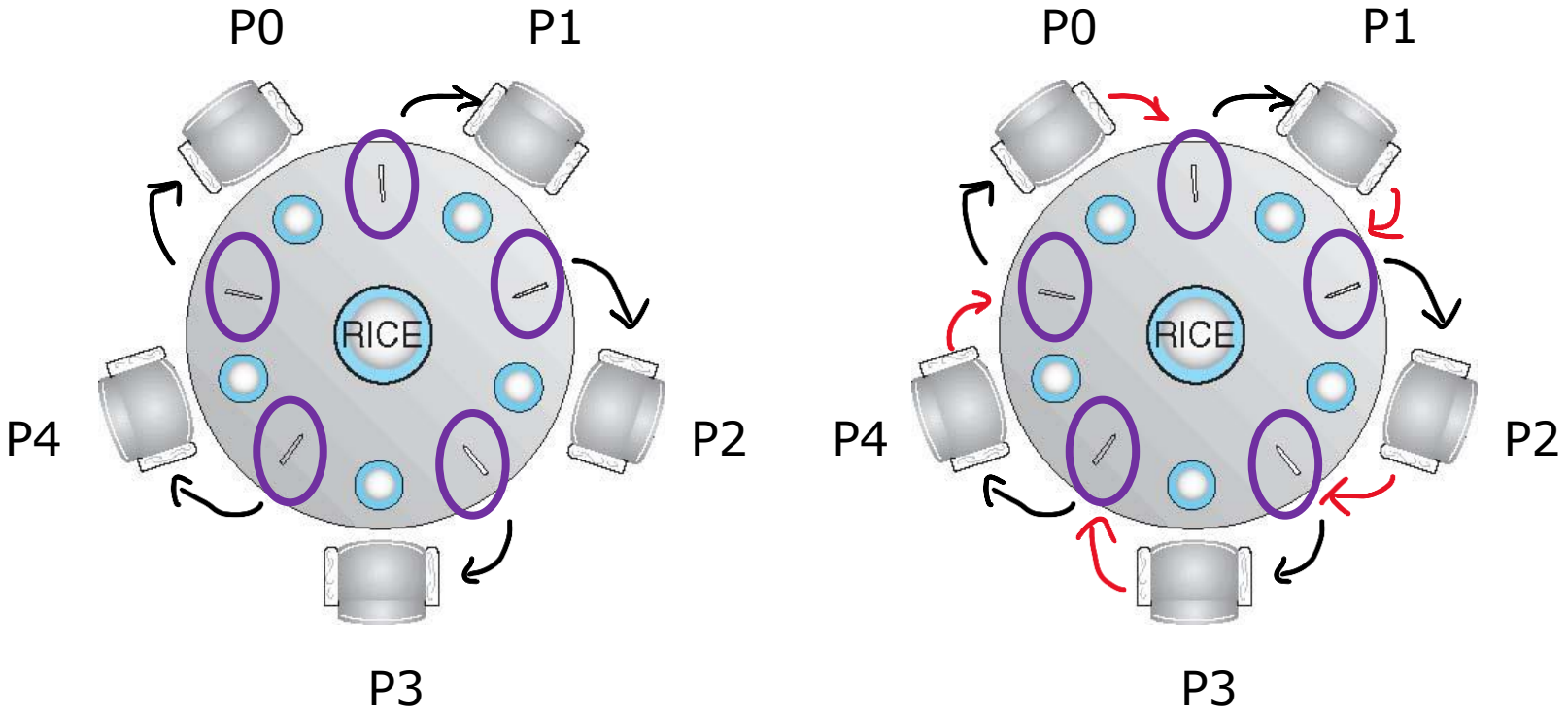
- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption of resources:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

circle process → waiting for each other → resource is held by others

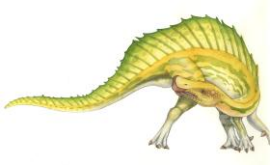
Necessary conditions for deadlock, not sufficient



When Can Deadlock Arise?



- 5 resource types (each chopstick)
- One instance of each



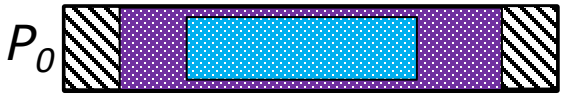
Deadlock with Mutex Locks

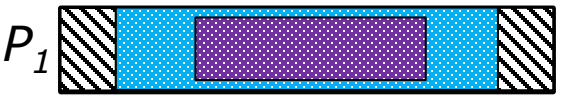
- Deadlocks can occur via system calls, locking, etc
- Example with Pthread

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);

   /* Do some work */

   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
```

preemption

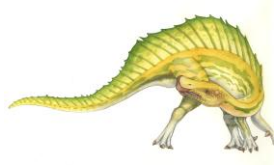
```
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);

   /* Do some work */

   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```





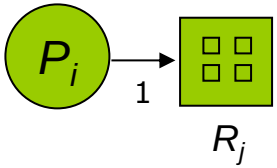
Resource-Allocation Graph

- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$
- Represents a specific state, it's a picture of a specific moment
  - Does not represent a flow or a behaviour

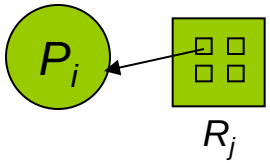


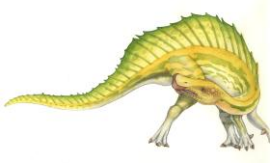
Resource-Allocation Graph (Cont.)

- Process 
- Resource Type with 4 instances 
- P_i requests 1 instance of R_j

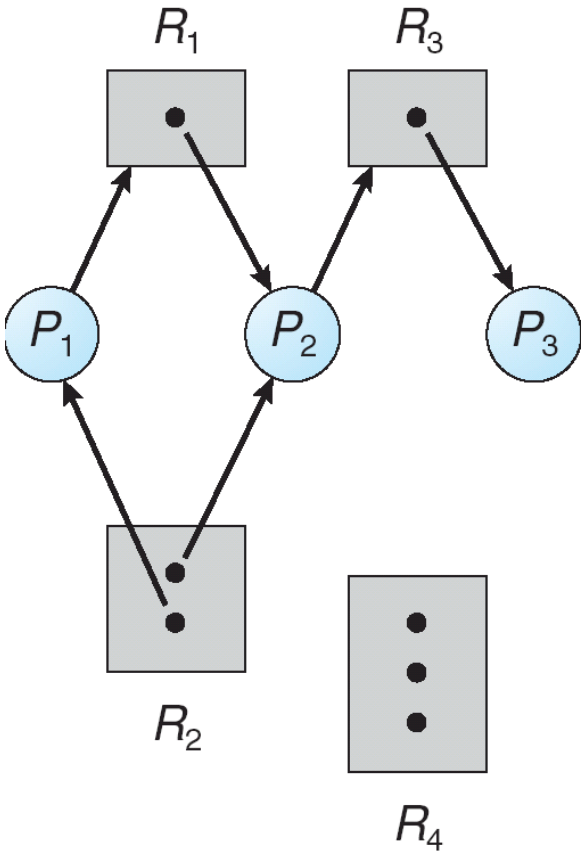


- P_i is holding an instance of R_j



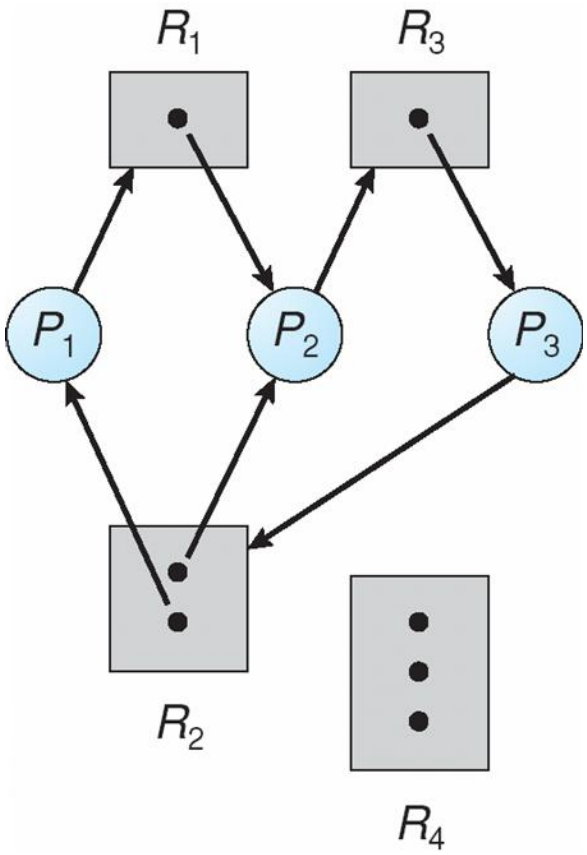


Example of a Resource Allocation Graph



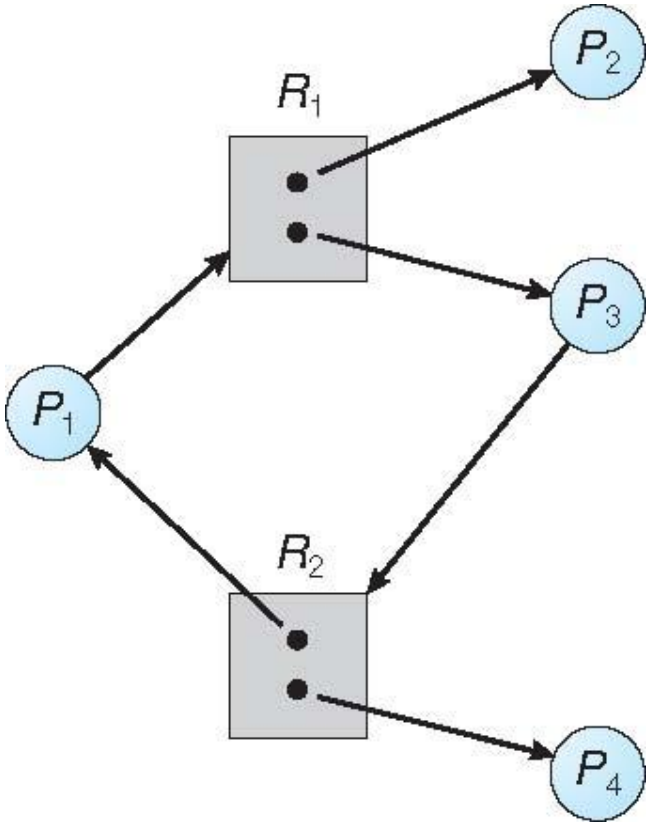
- Is there a deadlock?

Another Example of a Resource Allocation Graph

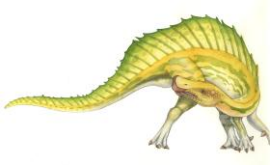


- Is there a deadlock?
- Is there any process holding all the resources it needs?

Yet Another Example of a Resource Allocation Graph

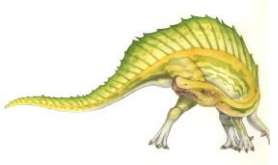


- Is there a deadlock?



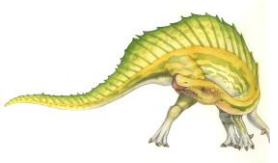
Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock



Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state (2 ways)
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state, determine if the system is in a deadlock, and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX. The user will have to deal with that.

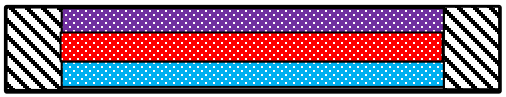


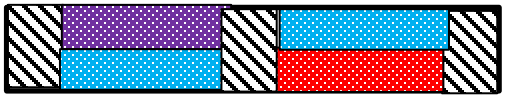
Deadlock Prevention

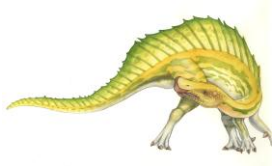
Restrain the ways request can be made.

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources
  - Some resources are intrinsically nonsharable (e.g., shared data)

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated *all its resources* before it begins execution
  - Or allow process to request new resources (even more than one at time) only when the process has none
  - Low resource utilization; starvation possible (popular resources)
    - Example: copy from DVD to disk; sort; print results

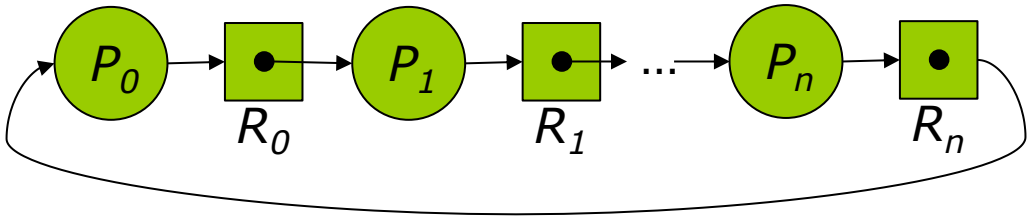


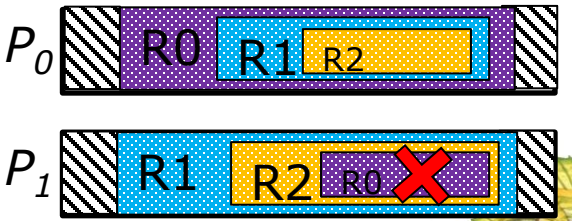




Deadlock Prevention (Cont.)

- **No Preemption of resources** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
    - possible problems: e.g.. data inconsistencies for shared memory
  - Alternative: process can "steal" resource from another one who is waiting
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration. ➔ inefficiencies may rise

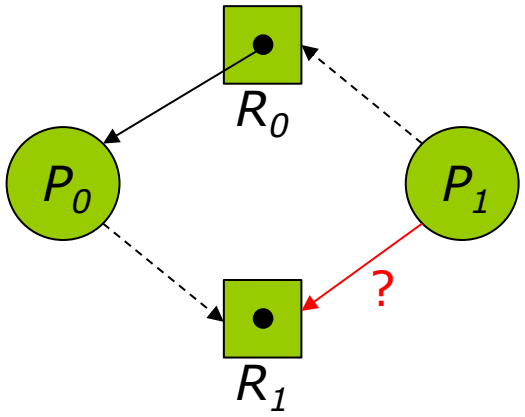


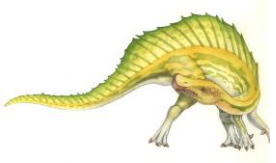


Deadlock Avoidance

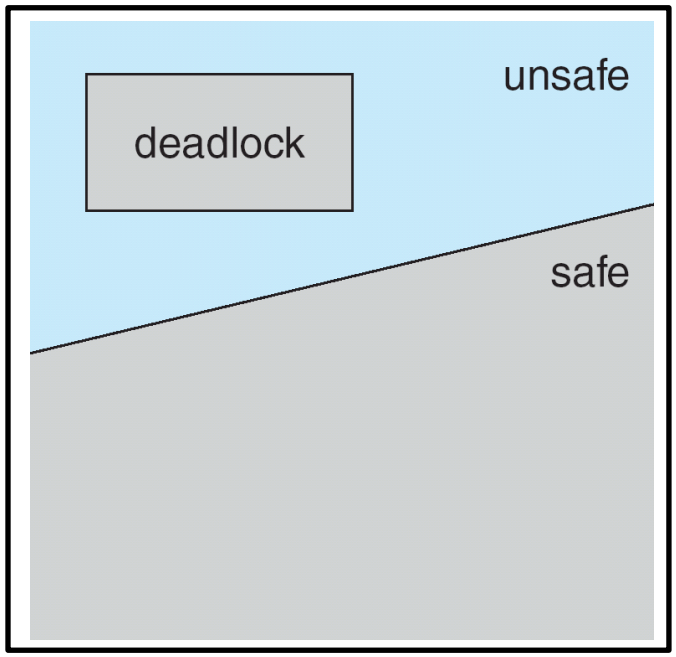
Requires that the system has some additional ***a priori*** information available.

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- **Resource-allocation state** is defined by the **number of available** and **allocated resources**, and the **maximum demands** of the processes





Safe, Unsafe, Deadlock State



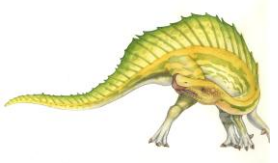
All possible resource-allocation states

We must know:

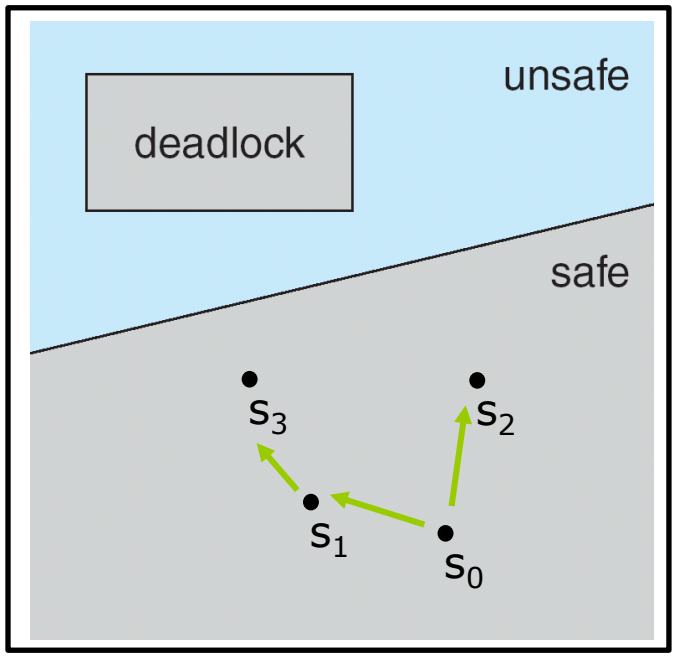
- for each process what is the **maximum demand** of resources it can issue

Each state is defined by:

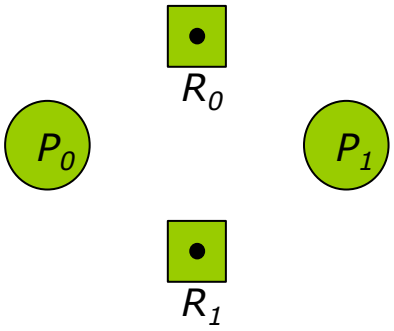
- **availability** of resources
- **allocation:** which resource is given to which process
- **requests:** which requests have been made by which process



Safe, Unsafe, Deadlock State



All possible resource-allocation states



Max demand

	R_0	R_1
P_0	1	1
P_1	1	1

s_0

		R_0	R_1
Avail.		1	1
Alloc.	P_0	0	0
	P_1	0	0
Req.	P_0	0	0
	P_1	0	0

s_1

R_0	R_1
1	1
0	0
0	0
1	0
0	0

s_3

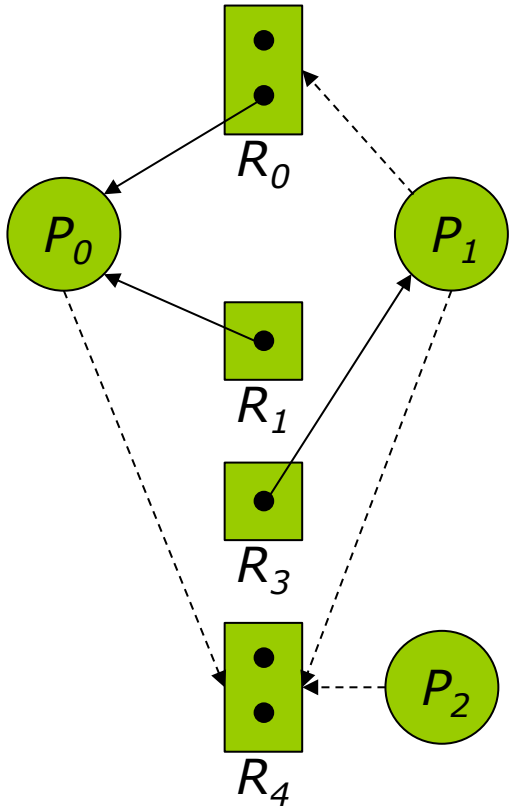
R_0	R_1
0	1
1	0
0	0
0	0
0	0

s_2

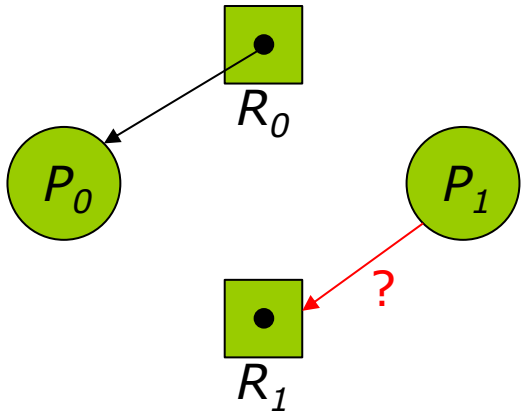
R_0	R_1
1	1
0	0
0	0
0	0
0	1

Safe State idea

- a safe state is a state in which for each process that is currently holding some resource I can guarantee that it can terminate even if it requests all the resources it can demand
  - must know the worst case: what are all the resources that a processes may request?
  - then, I decide if (even in the worst case) it can obtain the resources and finish.
- If that's the case for all processes currently holding some resources, then the OS can guide the scheduling in such a way that a process obtains all the resources that it needs (in the worst case) and finishes
- ➔ a safe state is a state in which there is a **safe sequence**: a possible scheduling by the OS that, even in the worst case, let processes terminate and free the resources, and let other processes continue.



Basic example



R_0 allocated to P_0
and P_1 requesting R_1:

		R_0	R_1
Avail.		0	1
Max dem.	P_0	1	1
	P_1	1	1
Alloc.	P_0	1	0
	P_1	0	0
Req.	P_0	0	0
	P_1	0	1

safe to grant the resource?

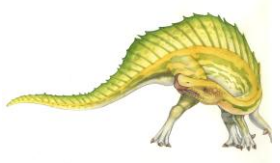
Need
(Max dem.–Alloc.)

	R_0	R_1
	0	0
P_0	0	1
P_1	1	0
P_0	1	0
P_1	0	1
P_0	0	0
P_1	0	0

- Is this a safe state?
- Can we define a safe sequence?
- Is there a process to which I can give all the resources it needs (worst possible case)?
- Is Avail. ≥ Need of any process?
- What about the previous state?

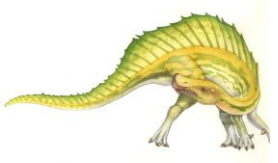
Safe State Definition

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When all P_j are finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on



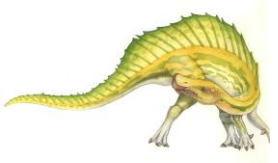
Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.



Deadlock Avoidance: Banker's Algorithm

- System model and assumptions:
  - Resource types with possibly multiple instances
  - Each process must a priori claim maximum use
  - When a process requests a resource, it may have to wait
  - When a process gets all its resources, it must return them in a finite amount of time



Example: check if state is safe

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5 instances), and *C* (7 instances)

Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Allocation	*Max*	*Available*		*Need*
	A B C	*A B C*	*A B C*		*A B C*
P_0	0 1 0	7 5 3	3 3 2	P_0	7 4 3
P_1	2 0 0	3 2 2		P_1	1 2 2
P_2	3 0 2	9 0 2		P_2	6 0 0
P_3	2 1 1	2 2 2		P_3	0 1 1
P_4	0 0 2	4 3 3		P_4	4 3 1

- Compute a safe sequence:
  1) Find a process for which Available>=Need
  2) The process can execute and terminate
  3) Release its resources (from Allocation to available)
  4) Repeat from 1

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true
  - It can be granted!

- Can it be SAFELY granted? Check if the new state is safe
  - New state:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Exercise (January 2016) Sol.

Consider the snapshot of a system in Table 1:

Table 1: Snapshot of resource status and requests.

	Allocation	*Need*	*Request*
	A B C D	A B C D	A B C D
P_1	0 0 0 1	2 2 0 1	2 2 0 0
P_2	1 0 0 0	0 2 2 2	0 1 2 0
P_3	1 3 5 4	2 0 0 2	1 0 0 1
P_4	0 0 1 2	0 0 0 2	0 0 0 1

Available: 1 2 2 0

E1.1) Is the system in a deadlock?

E1.2) Is the system in a safe state?

E1.3) Is there a request that can be safely granted? If so, which one?

- Is Avail. ≥ Need of any process holding some resource?

Exercise (May 2015) Sol.

Consider the following snapshot of a system:

	Allocation	*Max*	*Request*
	A B C D	A B C D	A B C D
P_1	0 0 1 2	0 0 2 3	0 0 0 1
P_2	0 1 0 0	2 1 2 0	1 0 1 0
P_3	3 1 5 4	3 2 5 6	0 1 0 0
P_4	0 0 0 1	2 2 0 1	2 2 0 0

Available
2 1 2 0

E2.1) Is the system in deadlock?

E2.2) Is the system in a safe state?

E2.3) Can P_3's request be safely granted immediately?

E2.4) If P_3's request is granted immediately, does the system enter a deadlock?

Be sure to motivate your answers.

Need (Max –Alloc.)

	A B C D
P_1	0 0 1 1
P_2	2 0 2 0
P_3	0 1 0 2
P_4	2 2 0 0

- Is Avail. ≥ Need of any process?

	Available
	2 1 2 0
P_2	2 2 2 0
P_4	2 2 2 1
P_1	2 2 3 3
P_3	.

A **safe sequence** exists

Exercise (May 2015)

Consider the following snapshot of a system:

	Allocation	*Max*	*Request*
	A B C D	A B C D	A B C D
P_1	0 0 1 2	0 0 2 3	0 0 0 1
P_2	0 1 0 0	2 1 2 0	1 0 1 0
P_3	3 2 1 5 4	3 2 5 6	0 0 1 0 0
P_4	0 0 0 1	2 2 0 1	2 2 0 0

Available: 2 1 2 0

E2.1) Is the system in deadlock?

E2.2) Is the system in a safe state? ➔YES

E2.3) Can P_3's request be safely granted immediately?

E2.4) If P_3's request is granted immediately, does the system enter a deadlock?

Be sure to motivate your answers.

- If P3's request is granted, system transit to a different state. Is it safe?

Need (Max –Alloc.)

	A B C D
P_1	0 0 1 1
P_2	2 0 2 0
P_3	0 1 0 2
P_4	2 2 0 0

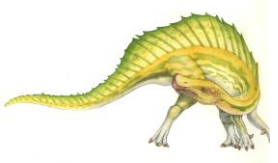
	Available
	2 0 2 0
P_2	2 1 2 0

X

➔ ∄ safe sequence

Deadlock Detection

- Necessary for the recovery strategy: detect and then recover
- Allow system to enter deadlock state
- Detection algorithm:
  - determine if the system can continue in the best possible case,
    - i.e., determine if (al least) an **execution sequence** exists



Example of Detection

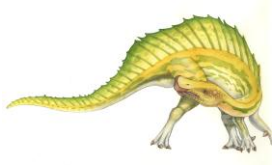
- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Find a process who can continue (in the best possible case) given the available resources at this stage
- i.e., a process for which Avail. ≥ Request

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***



Example (Cont.)

- What if P_2 requests an additional instance of type *C*?

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 **1**	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes' requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Exercise (January 2016)

Consider the snapshot of a system in Table 1:

Table 1: Snapshot of resource status and requests.

	Allocation	*Need*	*Request*
	A B C D	A B C D	A B C D
P_1	0 0 0 1	2 2 0 1	2 2 0 0
P_2	1 0 0 0	0 2 2 2	0 1 2 0
P_3	1 3 5 4	2 0 0 2	1 0 0 1
P_4	0 0 1 2	0 0 0 2	0 0 0 1

Available: 1 2 2 0

E1.1) Is the system in a deadlock?

E1.2) Is the system in a safe state? ➔**NO**

E1.3) Is there a request that can be safely granted? If so, which one?

- Does a process exist for which Avail. ≥ Request ?

Available

1 2 2 0

P_2 2 2 2 0

P_1 2 2 2 1

P_3 *or* P_4

An **execution sequence** exists

➔ no deadlock

Exercise (May 2015)

Consider the following snapshot of a system:

	Allocation	*Max*	*Request*
	A B C D	A B C D	A B C D
P_1	0 0 1 2	0 0 2 3	0 0 0 1
P_2	0 1 0 0	2 1 2 0	1 0 1 0
P_3	3 1 5 4	3 2 5 6	0 1 0 0
P_4	0 0 0 1	2 2 0 1	2 2 0 0

Available
2 1 2 0

E2.1) Is the system in deadlock?

E2.2) Is the system in a safe state? ➔**YES**

E2.3) Can P_3's request be safely granted immediately? ➔**NO**

E2.4) If P_3's request is granted immediately, does the system enter a deadlock?

Be sure to motivate your answers.

Exercise (May 2015)

Consider the following snapshot of a system:

	Allocation	*Max*	*Request*
	A B C D	A B C D	A B C D
P_1	0 0 1 2	0 0 2 3	0 0 0 1
P_2	0 1 0 0	2 1 2 0	1 0 1 0
P_3	3 ~~2~~ 1 5 4	3 2 5 6	0 ~~0~~ 1 0 0
P_4	0 0 0 1	2 2 0 1	2 2 0 0

Available: 2 ~~0~~ 1 2 0

- If P3's request is granted, system transit to a different state. Is it such that a process exists for which Avail.≥Request?

Need (Max –Alloc.)

	A B C D
P_1	0 0 1 1
P_2	2 0 2 0
P_3	0 ~~1~~ 0 1 0 2
P_4	2 2 0 0

E2.1) Is the system in deadlock? ➔**NO**

E2.2) Is the system in a safe state? ➔**YES**

E2.3) Can P_3's request be safely granted immediately? ➔**NO**

E2.4) If P_3's request is granted immediately, does the system enter a deadlock?

Be sure to motivate your answers.

Available
2 0 2 0
P_3 5 2 7 4
P_1 or P_2 or P_4

An **execution sequence** exists ➔ no deadlock

Quiz

A. What Tables are needed to check if the system is in a safe state?

B. What Tables are needed to check if the system is in a deadlock?

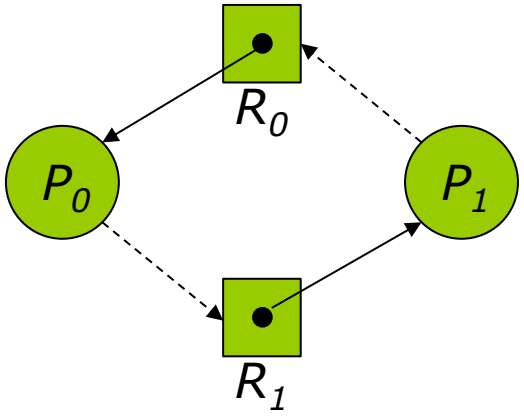
C. If the system is not in a safe state, there is a deadlock

D. If the system is in a deadlock, then it's not in a safe state

E. If a request can be safely granted, then it does not cause a deadlock

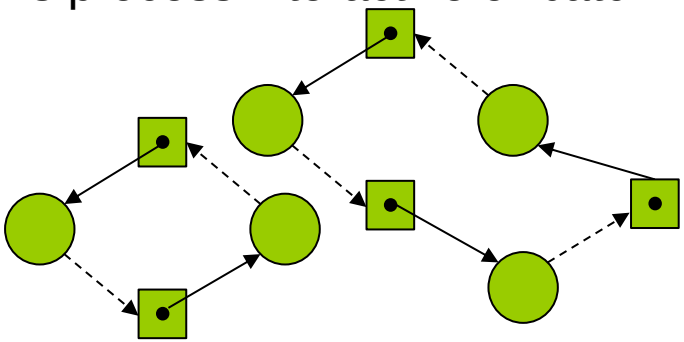
Detection-Algorithm Usage

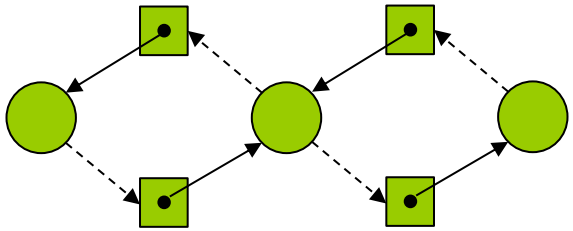
- Detection can be either done:
  - periodically
  - by monitoring the CPU cycles
- If deadlock is detected, recovery
  - cannot take the resources back from a process. That would leave the critical section unfinished.
  - must select a process to terminate

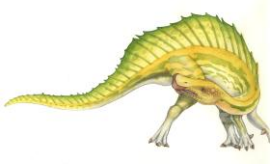


Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

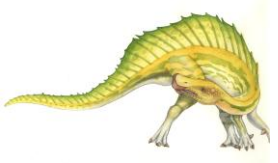






Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process from that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor



Exercise (book, page 340)

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

a. What is the content of the matrix *Need*?

b. Is the system in a safe state?

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately?

d. Can it be granted safely safe?

e. Does it create a deadlock?

Exercise (book, page 340)

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Need (Max –Alloc.)

	A B C D
P_0	0 0 0 0
P_1	0 7 5 0
P_2	1 0 0 2
P_3	0 0 2 0
P_3	0 6 4 2

a. What is the content of the matrix *Need*?

b. Is the system in a safe state?

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately?

d. Can it be granted safely?

e. Does it create a deadlock?

Exercise (book, page 340)

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Need (Max –Alloc.)

	A B C D
P_0	0 0 0 0
P_1	0 7 5 0
P_2	1 0 0 2
P_3	0 0 2 0
P_3	0 6 4 2

Available

A B C D

1 5 2 0

P_0 1 5 3 2

P_2 2 8 8 6

… (any order)

a. What is the content of the matrix *Need*?

b. Is the system in a safe state? **YES** Found a safe sequence!

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately?

d. Can it be granted safely?

e. Does it create a deadlock?

Exercise (book, page 340)

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	1 1 0 0
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	
P1	1 4 2 0		

Need (Max –Alloc.)

	A B C D
P_0	0 0 0 0
P_1	0 7 5 0
P_2	1 0 0 2
P_3	0 0 2 0
P_3	0 6 4 2

Available

	A B C D
	1 5 2 0
P_0	1 5 3 2
P_2	2 8 8 6

… (any order)

a. What is the content of the matrix *Need*?

b. Is the system in a safe state? **YES**

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately? **YES**

Available resources ≥ Requested resources

d. Can it be granted safely?

e. Does it create a deadlock?

Exercise (book, page 340)

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 1 0 0
P1	1 4 2 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Need (Max –Alloc.)

	A B C D
P_0	0 0 0 0
P_1	0 3 3 0
P_2	1 0 0 2
P_3	0 0 2 0
P_3	0 6 4 2

Available

	A B C D
	1 1 0 0
P_0	1 1 1 2
P_2	2 4 6 6
P_1	3 8 8 6

… (any order)

a. What is the content of the matrix *Need*?

b. Is the system in a safe state? **YES**

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately? **YES**

d. Can it be granted safely? **YES** Update the NEED table and find a new SAFE sequence

e. Does it create a deadlock?

Exercise (book, page 340)

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 1 0 0
P1	1 4 2 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Need (Max –Alloc.)

	A B C D
P_0	0 0 0 0
P_1	0 3 3 0
P_2	1 0 0 2
P_3	0 0 2 0
P_3	0 6 4 2

Available

A B C D
1 1 0 0
P_0 1 1 1 2
P_2 2 4 6 6
P_1 3 8 8 6
… (any order)

a. What is the content of the matrix *Need*?

b. Is the system in a safe state? **YES**

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately? **YES**

d. Can it be granted safely? **YES**

e. Does it create a deadlock? **NO**

If a request can be safely grante, it NEVER creates a deadlock