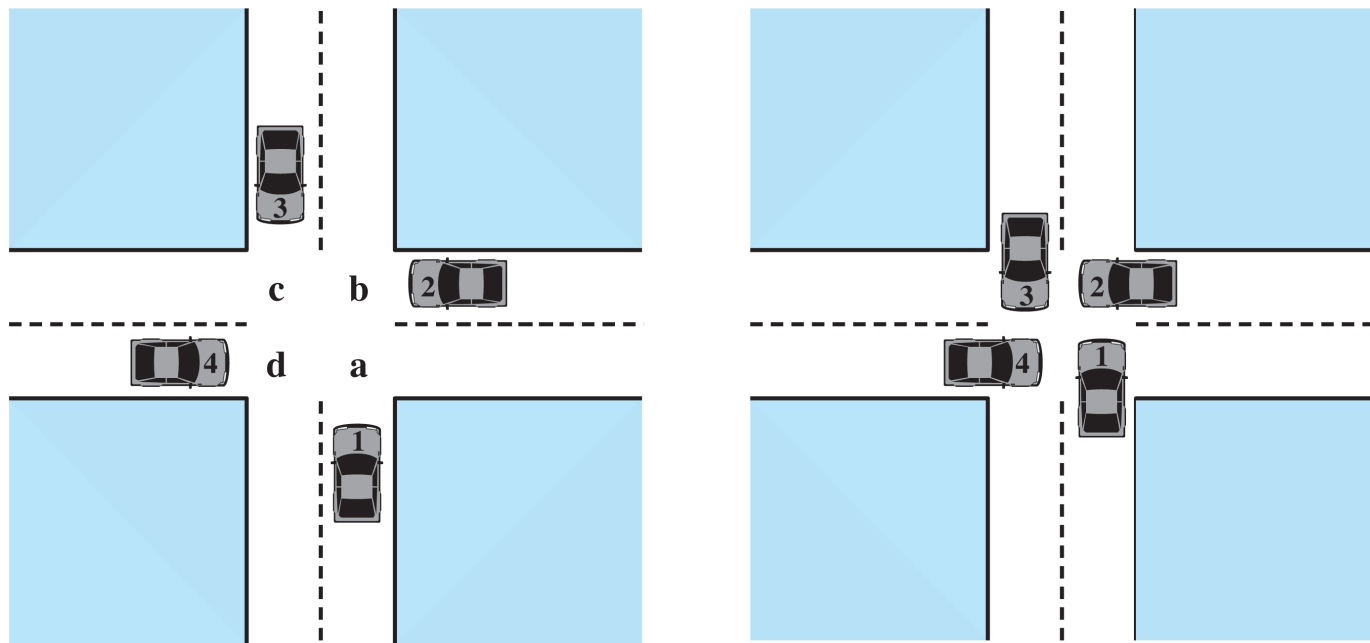
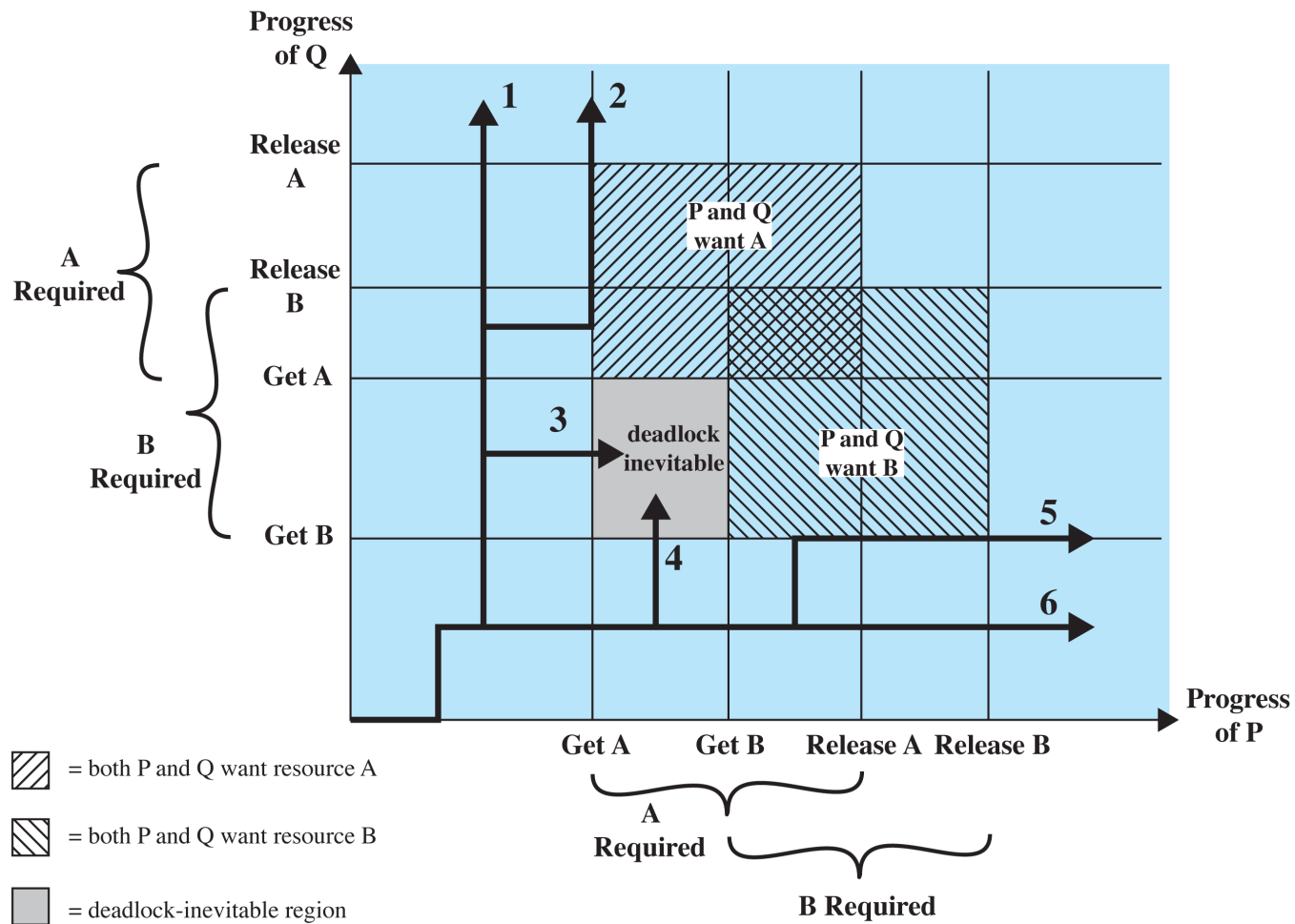


DEADLOCK

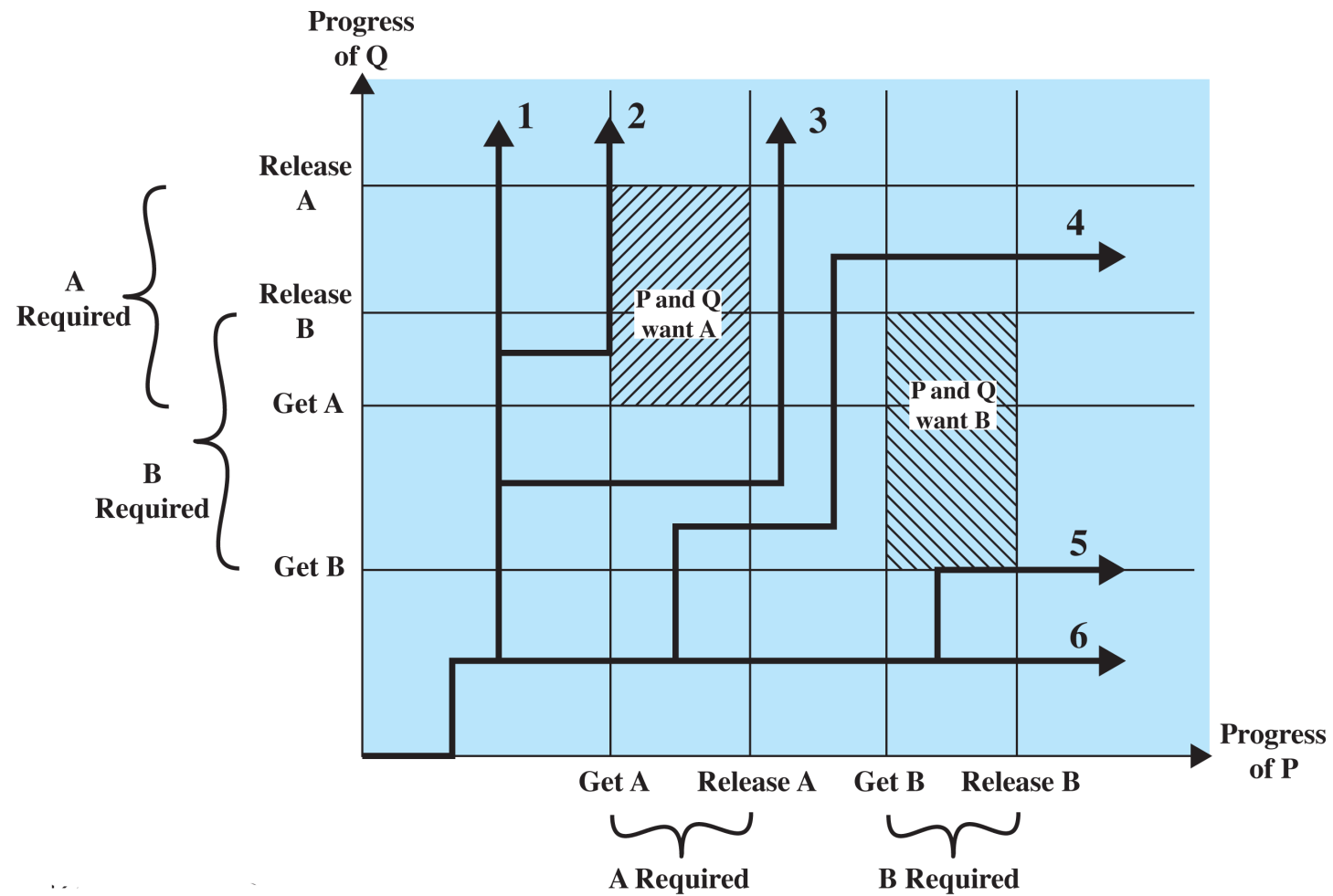
- The permanent blocking of a set of processes that either compete for resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent; no efficient solution



JOINT PROGRESS WITH DEADLOCK



CS 409, FALL 2013



RESOURCE CATEGORIES AND DEADLOCK

- **Reusable** = safely used by only one process at a time but not depleted by that use
 - Processors, I/O channels, memory, devices, files, databases, semaphores
 - Sample deadlocks:

P1:	P2:	(200 KB memory available overall)	
lock (&l2);	lock (&l1);		
...	...		
lock (&l1);	lock (&l2);	P1:	P2:
...	...	Request 80 KB	Request 70 KB
unlock (&l2);	unlock (&l2);
unlock (&l1);	unlock (&l1);	Request 60 KB	Request 80 KB
	

- **Consumable** = can be created (produced) and destroyed (consumed)
 - interrupts, signals, messages, information, data in I/O buffers
 - Sample deadlock (receive blocking):

P1:	P2:
Receive (P2)	Receive (P1)
...	...
Send (P2, M1)	Send (P1, M2)
...	...

CONDITIONS FOR DEADLOCK

- **Mutual exclusion**
- **Hold-and-wait** (a process may hold allocated resources while awaiting assignment of others)
- **No preemption** (no resource can be forcibly removed from a process holding it)
- **Circular wait** (a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain)
- Dealing with deadlock conditions:
 - **Prevent deadlock** (adopt a policy that eliminates one of the conditions)
 - **Avoid deadlock** (make the appropriate dynamic choices based on the current state of resource allocation)
 - **Detect deadlock** (attempt to detect the presence of deadlock and take action to recover)

APPROACHES TO DEADLOCK DETECTION, PREVENTION, AND AVOIDANCE

- **Prevention**: conservative, undercommits resources
 - **Requesting all resources at once**
Advantages: best for processes that perform a single burst of activity; no preemption necessary
Disadvantages: inefficient; delays process initiation; requirements must be known in advance
 - **Preemption**
Advantages: convenient for resources whose state can be saved and restored easily
Disadvantages: preempts more often than necessary
 - **Resource ordering**
Advantages: enforceable via compile-time checks, so needs no run-time computation
Disadvantages: disallows incremental resource requests
- **Avoidance**: finds at least one safe path; midway between detection and prevention
Advantages: no preemption necessary
Disadvantages: future resource requirements must be known; processes blocked for long periods
- **Detection**: requested resources are granted where possible (very liberal); must be invoked periodically to test for deadlock
Advantages: never delays process initiation; facilitates online handling
Disadvantages: inherent preemption losses

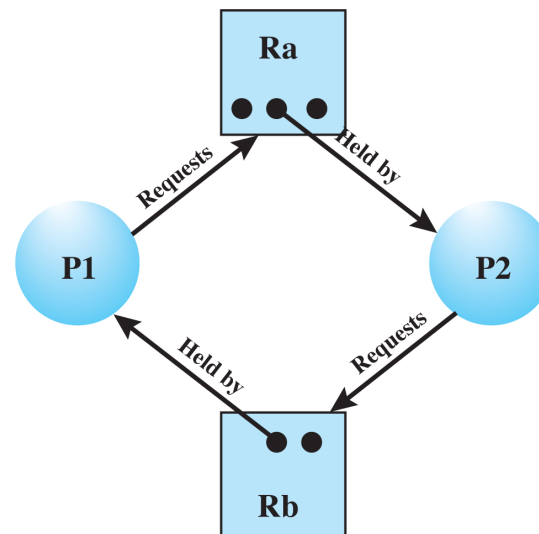
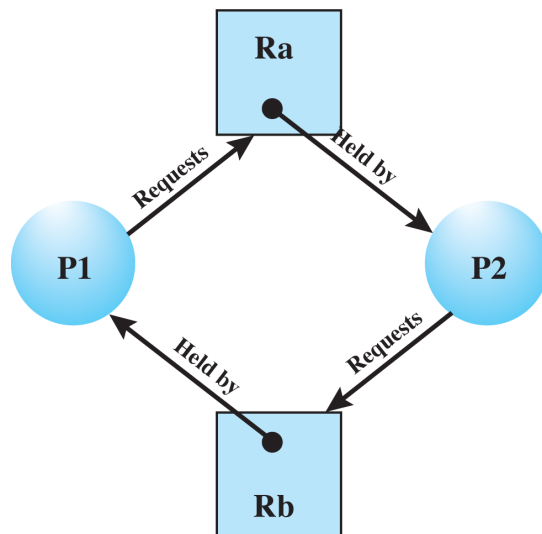
RESOURCE ALLOCATION GRAPHS



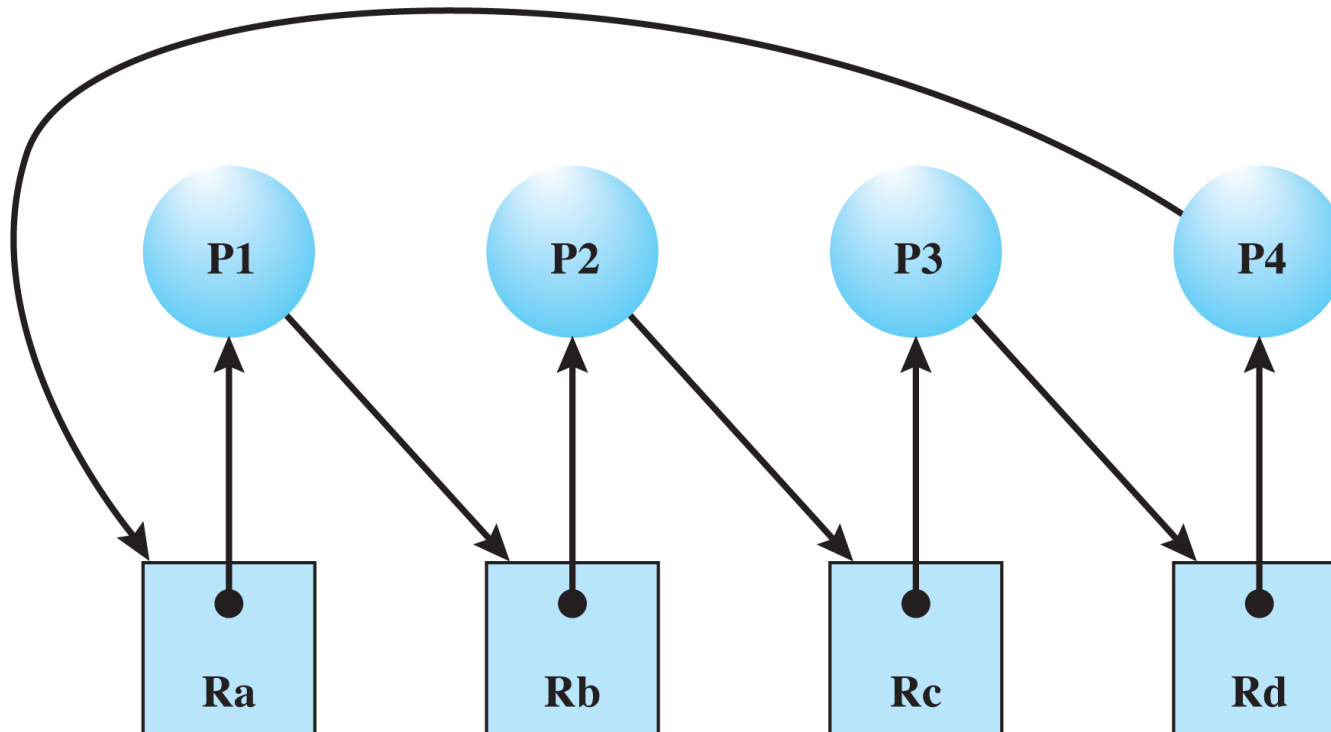
(a) Resource is requested



(b) Resource is held



RESOURCE ALLOCATION GRAPHS (CONT'D)



DEADLOCK PREVENTION

- Design a system in such a way that the possibility of deadlock is excluded
- Two main methods:
 - **Indirect** – prevent the occurrence of one of the three necessary conditions
 - * **Mutual exclusion**: not required for sharable resources (but must hold for non-sharable resources)
 - * **Hold and wait**: require that a process request all of its required resources at one time and block the process until all requests can be granted simultaneously
 - * **No preemption**: if a process holding certain resources is denied a further request, that process must release its original resources and request them again
 - * **Circular wait**: define a linear ordering on resource types
 - **Direct** – prevent the occurrence of a circular wait

DEADLOCK AVOIDANCE

- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Requires knowledge of future process requests
- Approaches:
 - **Resource Allocation Denial**: do not grant an incremental resource request to a process if this allocation might lead to deadlock
 - **Process Initiation Denial**: do not start a process if its demands might lead to deadlock
- Algorithms:
 - Single instance of a resource type: use the **resource-allocation graph**
 - * The resource is granted iff granting it does not create a cycle
 - Multiple instances of a resource type: use the **banker's algorithm**

RESOURCE ALLOCATION DENIAL: THE BANKER'S ALGORITHM

- **State** of the system reflects the current allocation of resources to processes
- **Safe state** is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock; the opposite is an **unsafe state**

```
struct state{ int resource[m]; int available[m];
              int claim[n][m]; int alloc[n][m];

if (alloc[i, *] + request[*] > claim[i, *]) < error >; // tot. request > claim
else if (request[*] > available[*]) < suspend process >;
else {                                           // simulate alloc
    < define newstate as:
        alloc[i, *] = alloc[i, *] + request[*];
        available[*] = available[*] - request[*]; >;
    }
if (safe(newstate)) < carry out the allocation >;
else {
    < restore original state >;
    < suspend process >;
}
```

BANKER'S ALGORITHM (CONT'D)

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found) {                                /* simulate execution of Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

SAFE STATES

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

SAFE STATES (CONT'D)

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) **P1 runs to completion**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) **P3 runs to completion**

UNSAFE STATES

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

DEADLOCK AVOIDANCE (CONT'D)

- **Advantages**

- It is not necessary to preempt and rollback processes, as in deadlock detection
- It is less restrictive than deadlock prevention

- **Restrictions**

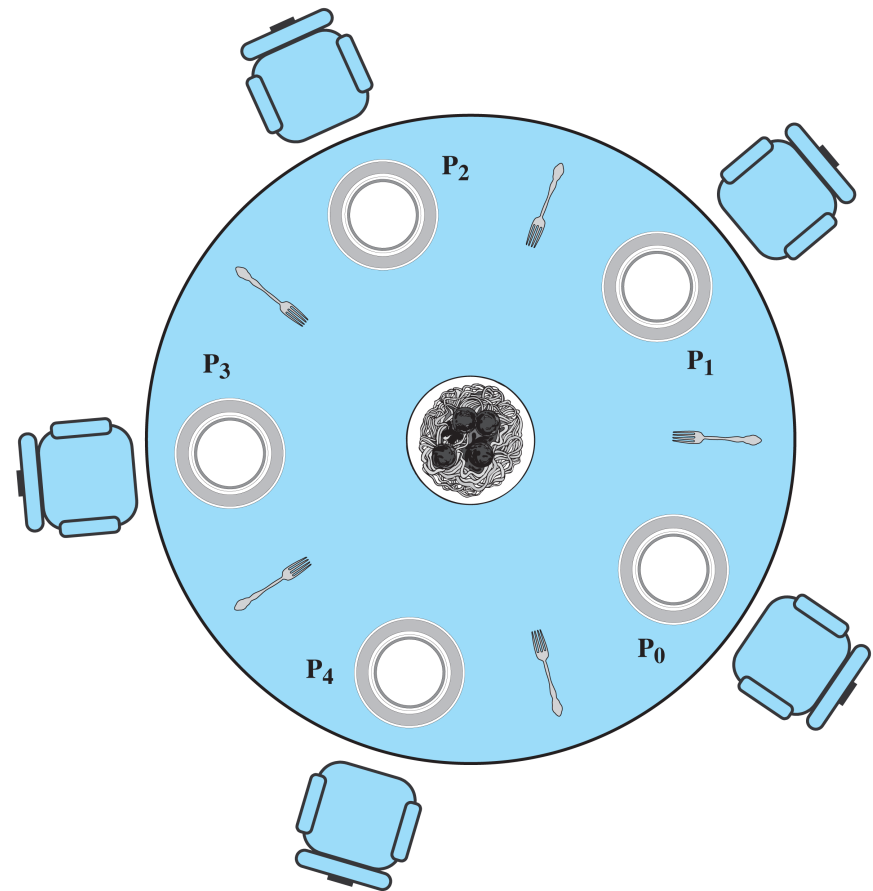
- Maximum resource requirement for each process must be stated in advance
- Processes under consideration must be independent and with no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

DEADLOCK DETECTION ALGORITHMS

- Deadlock prevention is conservative (limits access by restricting processes)
- By contrast deadlock detection strategies do the opposite: resource requests are granted whenever possible
 - Check for deadlock can be made as frequently as needed
- **Advantages:**
 - Early detection
 - Relatively simple algorithms
- **Disadvantage:**
 - Frequent checks consume considerable processor time
- **Recovery strategies** necessary
 - Abort all deadlocked processes
 - Back up deadlocked processes to some checkpoint and restart all processes
 - Successively abort deadlocked processes until deadlock no longer exists
 - Successively preempt resources until deadlock no longer exists

THE DINING PHILOSOPHERS PROBLEM

- **Mutual exclusion:** no two philosophers can use the same fork at the same time
- **Avoid deadlock and starvation:** no philosopher must starve to death



DINING PHILOSOPHERS WITH SEMAPHORES

```
semaphore fork [5] = {1, 1, 1, 1, 1};

void philosopher (int i) {
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}

void main() {
    for int i = 0 to 5 run in parallel philosopher(i);
}
```

BETTER DINING PHILOSOPHERS

```
semaphore fork[5] = {1, 1, 1, 1, 1};  
semaphore room = 4;
```

```
void philosopher (int i) {  
    while (true) {  
        think();  
        wait (room);  
        wait (fork[i]);  
        wait (fork [(i+1) mod 5]);  
        eat();  
        signal (fork [(i+1) mod 5]);  
        signal (fork[i]);  
        signal (room);  
    }  
}
```

```
void main() {  
    for int i = 0 to 5 run in parallel philosopher(i);  
}
```

DINING PHILOSOPHERS WITH A SEMAPHORE

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true, ...}; /* availability status of each fork */
void get_forks(int pid) {
    int left = pid;    int right = (++pid) % 5;
    if (!fork(left))           /*grant the left fork*/
        cwait(ForkReady[left]); /* queue on condition variable */
    fork(left) = false;
    if (!fork(right))          /*grant the right fork*/
        cwait(ForkReady[right]); /* queue on condition variable */
    fork(right) = false;
}
void release_forks(int pid) {
    int left = pid;    int right = (++pid) % 5;
    if (empty(ForkReady[left])) /*no one is waiting for this fork */
        fork(left) = true;      /*release the left fork*/
    else                      /* awaken a process waiting on this fork */
        csignal(ForkReady[left]);
    if (empty(ForkReady[right])) /*no one is waiting for this fork */
        fork(right) = true;     /*release the right fork*/
    else                      /* awaken a process waiting on this fork */
        csignal(ForkReady[right]);
}
```

DINING PHILOSOPHERS WITH A SEMAPHORE (CONT'D)

```
void philosopher[k=0 to 4] /* the five philosopher clients */
{
    while (true) {
        <think>;
        get_forks(k);          /* client requests two forks via monitor */
        <eat spaghetti>;
        release_forks(k);      /* client releases forks via the monitor */
    }
}
```

MOST COMMON OS APPROACH TO DEADLOCK

MOST COMMON OS APPROACH TO DEADLOCK

Ignore the problem and pretend that deadlocks
never occur in the system!

UNIX CONCURRENCY MECHANISMS

- **Pipes**: FIFO queues, implement the producer/consumer model
- **Messages**: `msgsnd` and `msgrcv` system calls with one message queue for each process
- **Signals**: primitive messages, used for signaling special conditions
- **Shared memory**:
 - Creation: `id = shmget(IPC_PRIVATE, size, S_IRUSR|S_IWUSR);`
 - Attach: `shared_memory = (char *) shmat(id, NULL, 0);`
 - Use: like a normal buffer (array)
 - Detach shared memory from own address space: `shmdt(shared_memory);`
 - Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory
- **Semaphores**: for mutual exclusion (as usual)

LINUX KERNEL CONCURRENCY MECHANISMS

- **Atomic Operations**: simplest approach to synchronization
 - Two types: **integer operations** (typical use: counters) and **bitmap operations**
- **Spinlocks**: most common technique for protecting a critical section in Linux
 - Integer location checked by each thread before it enters its critical section
 - Can only be acquired by one thread at a time; the others will keep trying (spinning) until they can acquire the lock
 - Effective whenever the wait time for acquiring a lock is expected to be very short
 - Disadvantage: busy-waiting
- **Semaphores**: binary, counting, readers/writers
- **Barriers**: enforce the order in which instructions are executed
 - `rmb ()` - prevents loads from being reordered across the barrier
 - `wmb ()` - prevents stores from being reordered across the barrier
 - `mb ()` - prevents both from being reordered across the barrier
 - etc.

LINUX ATOMIC OPERATIONS

Atomic integer operations

```
ATOMIC_INIT (int i)
int atomic_read(atomic_t *v)
void atomic_set(atomic_t *v, int i)
void atomic_add(int i, atomic_t *v)
void atomic_sub(int i, atomic_t *v)
void atomic_inc(atomic_t *v)
void atomic_dec(atomic_t *v)
int atomic_sub_and_test(int i, atomic_t *v)

int atomic_add_negative(int i, atomic_t *v)

int atomic_dec_and_test(atomic_t *v)

int atomic_inc_and_test(atomic_t *v)
```

At declaration: initialize an `atomic_t` to `i`
Read integer value of `v`
Set the value of `v` to integer `i`
Add `i` to `v`
Subtract `i` from `v`
Add 1 to `v`
Subtract 1 from `v`
Subtract `i` from `v`; return 1 if the result is zero; return 0 otherwise
Add `i` to `v`; return 1 if the result is negative; return 0 otherwise (used for semaphores)
Subtract 1 from `v`; return 1 if the result is zero; return 0 otherwise
Add 1 to `v`; return 1 if the result is zero; return 0 otherwise

Atomic bitmap operations

```
void set_bit(int nr, void *addr)
void clear_bit(int nr, void *addr)
void change_bit(int nr, void *addr)
int test_and_set_bit(int nr, void *addr)
int test_and_clear_bit(int nr, void *addr)
int test_and_change_bit(int nr, void *addr)
int test_bit(int nr, void *addr)
```

Set bit `nr` in the bitmap `addr`
Clear bit `nr` in the bitmap `addr`
Invert bit `nr`
Set bit `nr` and return the old bit value
Clear bit `nr` and return the old bit value
Invert bit `nr` and return the old bit value
Return the value of bit `nr`

LINUX SPINLOCKS

- `void spin_lock(spinlock_t *lock):` acquires the specified lock, spinning if needed until it is available
- `void spin_lock_irq(spinlock_t *lock):` also disables interrupts on the local processor
- `void spin_lock_irqsave(spinlock_t *lock, unsigned long flags):` also saves the current interrupt state in flags
- `void spin_unlock(spinlock_t *lock):` releases given lock
- `void spin_unlock_irq(spinlock_t *lock):` releases given lock and enables local interrupts
- `void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags):` releases given lock and restores local interrupts to given previous state
- `void spin_lock_init(spinlock_t *lock):` initializes given spinlock
- `int spin_trylock(spinlock_t *lock):` tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
- `int spin_is_locked(spinlock_t *lock):` returns nonzero if lock is currently held and zero otherwise

LINUX SEMAPHORES

- **Traditional semaphores:**

- `void sema_init(struct semaphore *sem, int count)`
- `void init_MUTEX(struct semaphore *sem)` (initially unlocked)
- `void init_MUTEX_LOCKED(struct semaphore *sem)` (initially locked)
- `void down(struct semaphore *sem)` (and enters uninterruptible sleep if semaphore is unavailable)
- `int down_interruptible(struct semaphore *sem)` (interruptible)
- `int down_trylock(struct semaphore *sem)`
- `void up(struct semaphore *sem)` (release semaphore)

- **Reader-writer semaphores:**

- `void init_rwsem(struct rw_semaphore, *rwsem)`
- `void down_read(struct rw_semaphore, *rwsem)`
- `void up_read(struct rw_semaphore, *rwsem)`
- `void down_write(struct rw_semaphore, *rwsem)`
- `void up_write(struct rw_semaphore, *rwsem)`