

# Sistem Operasi 2009

Pertemuan 6

## ***Concurrency: Deadlock & Starvation***

H u s n i

Lab. Sistem Komputer & Jaringan  
Teknik Informatika Univ. Trunojoyo

# Deadlock (1)

---

- *Permanent blocking* dari sekumpulan proses yang saling bersaing mendapatkan sumber daya sistem atau komunikasi
- Tidak ada solusi yang efisien
- Mencakup kebutuhan yang bertentangan bagi sumber daya oleh dua atau lebih proses

# Deadlock (2)

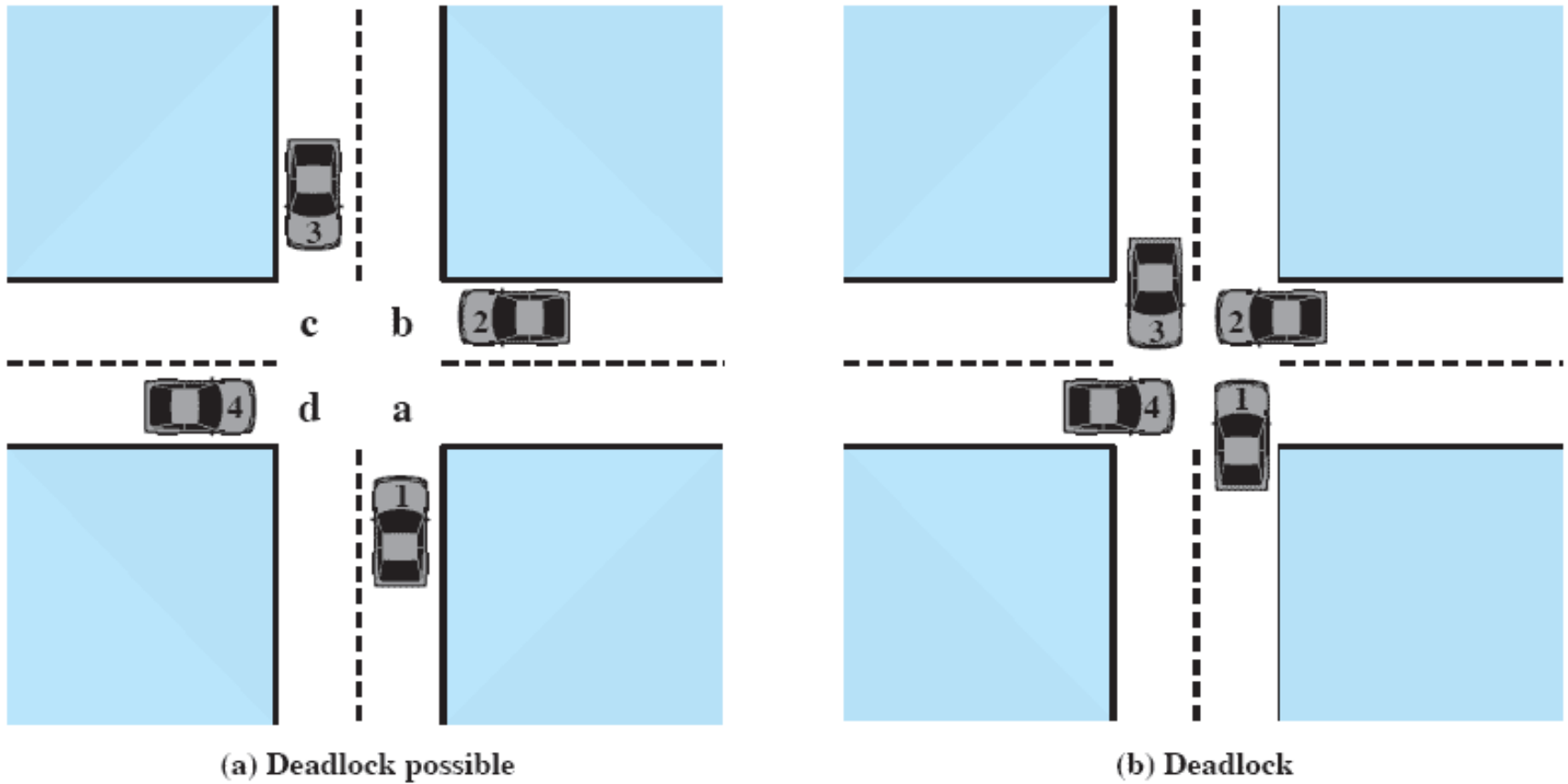


Figure 6.1 Illustration of Deadlock

# Deadlock (3)

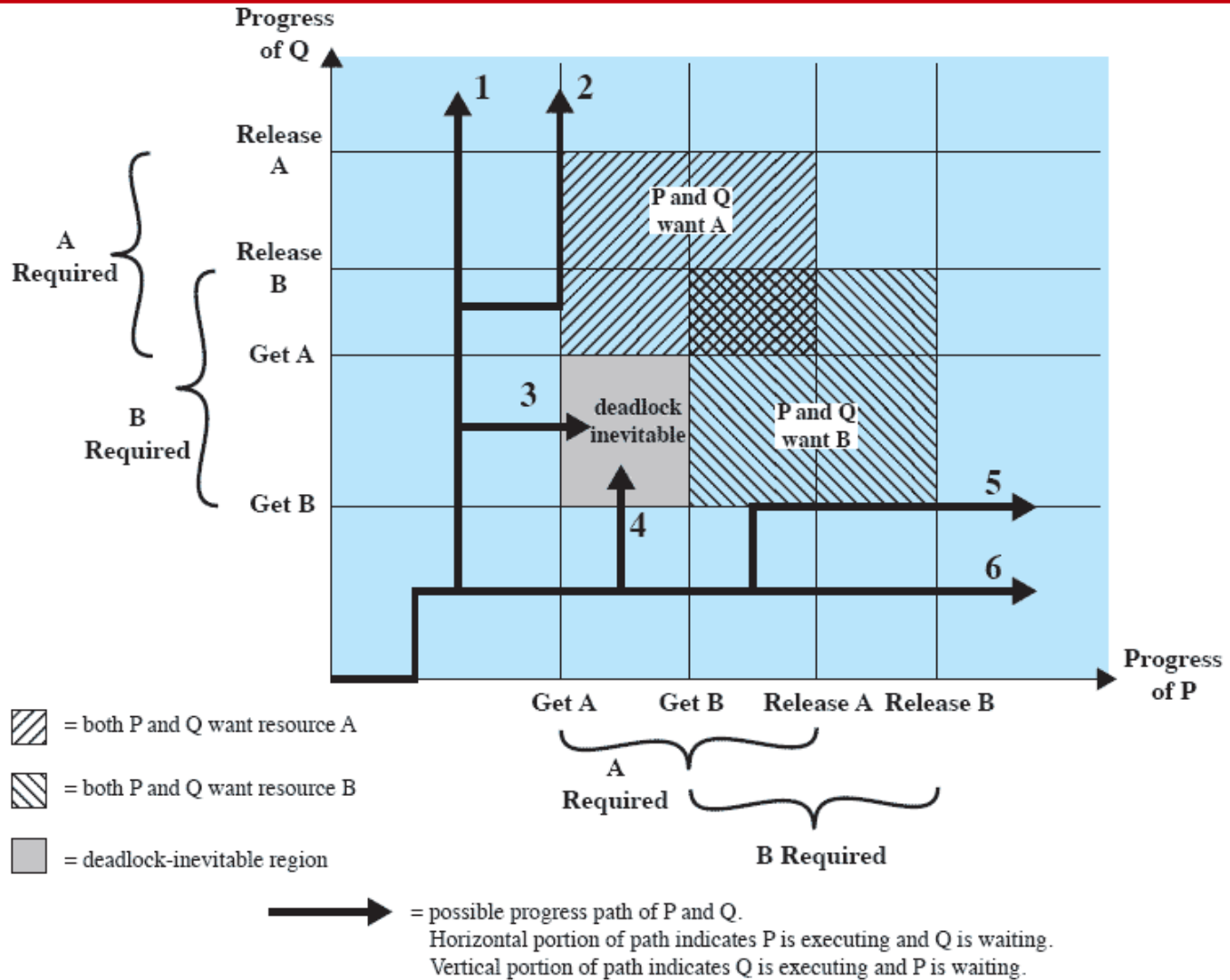


Figure 6.2 Example of Deadlock

# Deadlock (4)

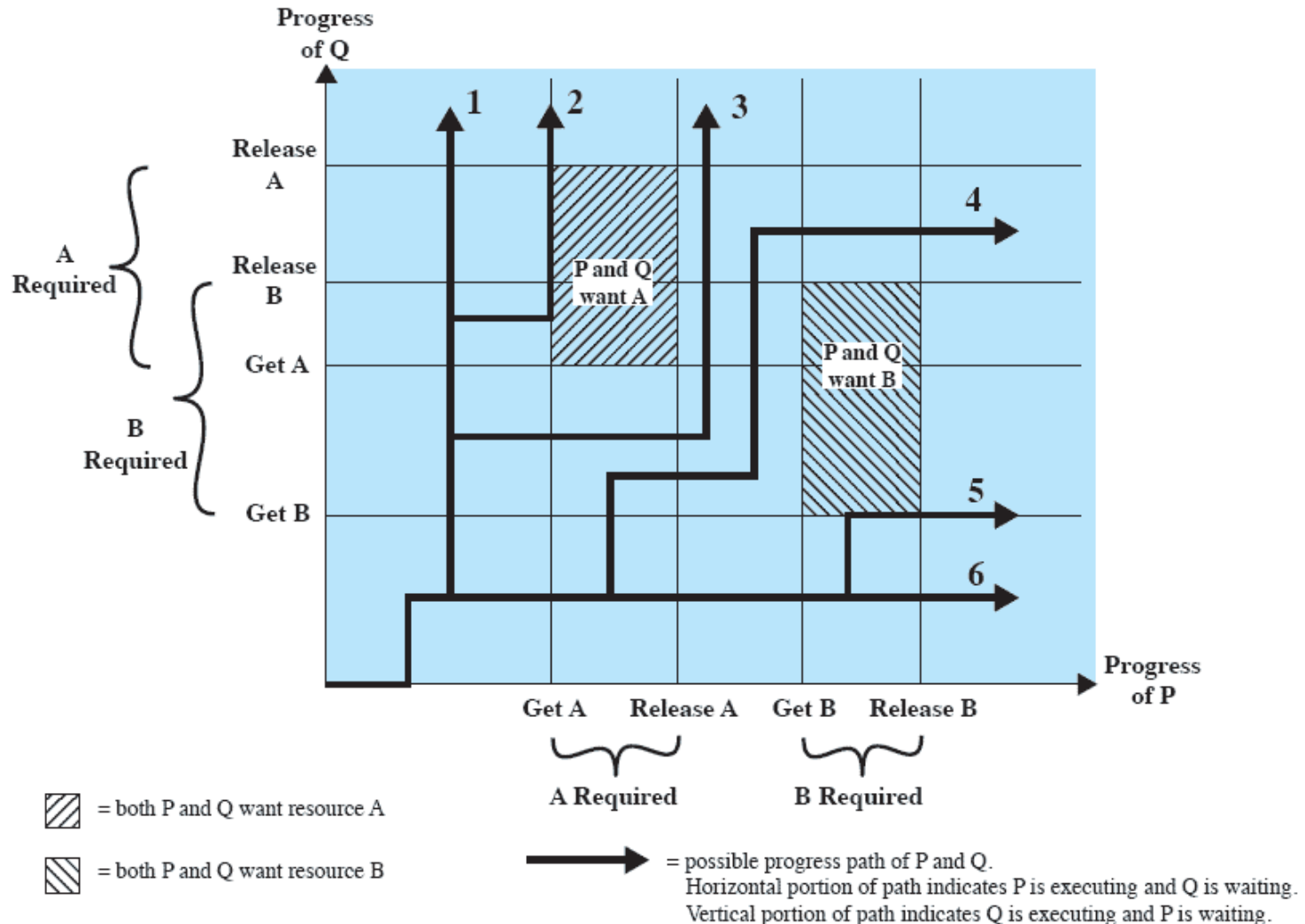


Figure 6.3 Example of No Deadlock [BACO03]

# Sumber Daya *Reusable* (1)

---

- Digunakan hanya oleh satu proses pada satu waktu dan tidak dihabiskan oleh penggunaan tersebut
- Proses memperoleh sumber daya, kemudian dilepaskan agar dapat digunakan (ulang, *reuse*) oleh proses lain

# Sumber Daya *Reusable* (2)

---

- Termasuk: Processor, Channel I/O, memory utama & sekunder, perangkat, dan struktur data seperti file, database, dan semaphore
- **Deadlock** terjadi jika setiap proses memegang satu sumber daya dan meminta sumber daya lain

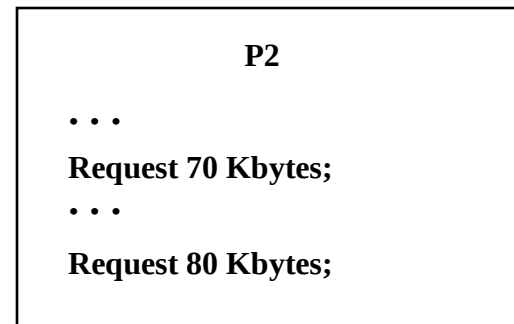
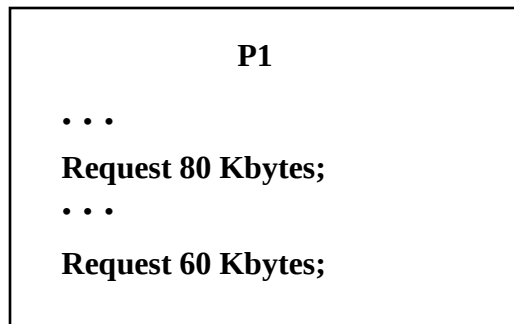
# Sumber Daya *Reusable* (3)

Process P		Process Q	
Step	Action	Step	Action
p <sub>0</sub>	Request (D)	q <sub>0</sub>	Request (T)
p <sub>1</sub>	Lock (D)	q <sub>1</sub>	Lock (T)
p <sub>2</sub>	Request (T)	q <sub>2</sub>	Request (D)
p <sub>3</sub>	Lock (T)	q <sub>3</sub>	Lock (D)
p <sub>4</sub>	Perform function	q <sub>4</sub>	Perform function
p <sub>5</sub>	Unlock (D)	q <sub>5</sub>	Unlock (T)
p <sub>6</sub>	Unlock (T)	q <sub>6</sub>	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

# Sumber Daya *Reusable* (4)

- Space (ruang di memory) tersedia bagi alokasi 200 KB, dan deretan kejadian berikut terjadi



- Deadlock terjadi jika kedua proses bergerak ke permintaan kedua mereka

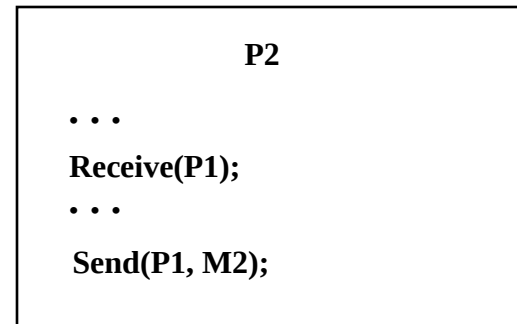
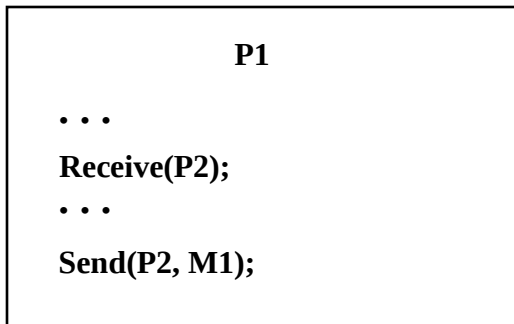
# Sumber Daya *Consumable*

---

- Dibuat (*produced*) dan dihancurkan (*consumed*)
- Termasuk: Interrupt, signal, message, dan informasi dalam buffer I/O
- Deadlock dapat terjadi jika suatu receive message di-blocking
- Dapat berupa kombinasi jarang dari *event-event* yang menyebabkan deadlock

# Contoh Deadlock

- **Deadlock** terjadi jika receive di-blocking



# Grafik Alokasi *Resource*

- *Directed graph* yang menggambarkan status dari sistem sumber daya dan proses



(a) Resource is requested



(b) Resource is held

# Kondisi Untuk Deadlock (1)

---

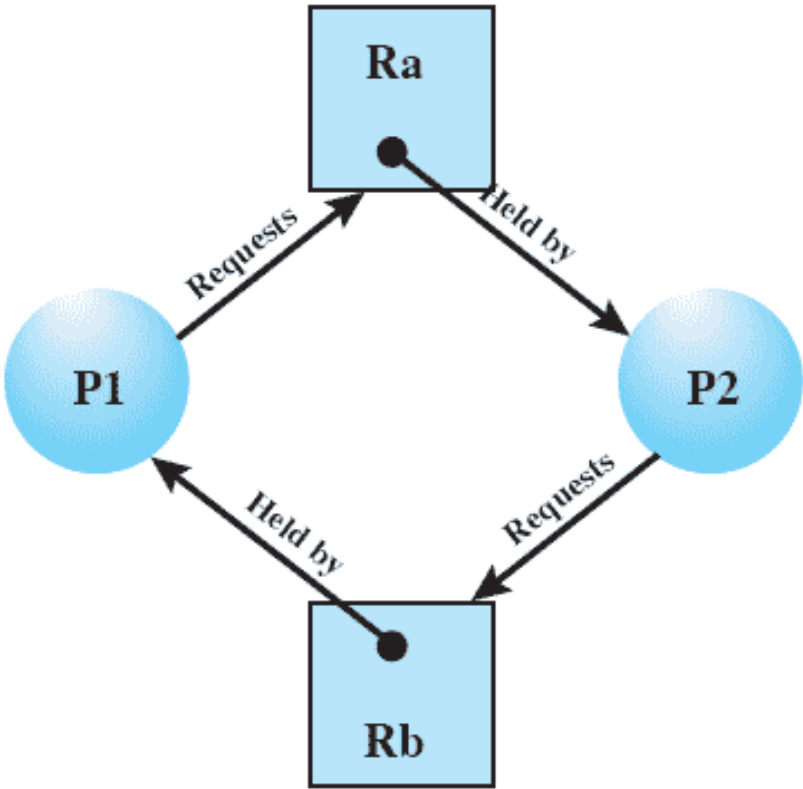
- Mutual exclusion
  - Hanya satu proses yang boleh menggunakan suatu sumber daya pada satu waktu
- Hold-and-wait
  - Suatu proses boleh memegang sumber daya yang dialokasikan selama menunggu *assignment* yang lain

# Kondisi Untuk Deadlock (2)

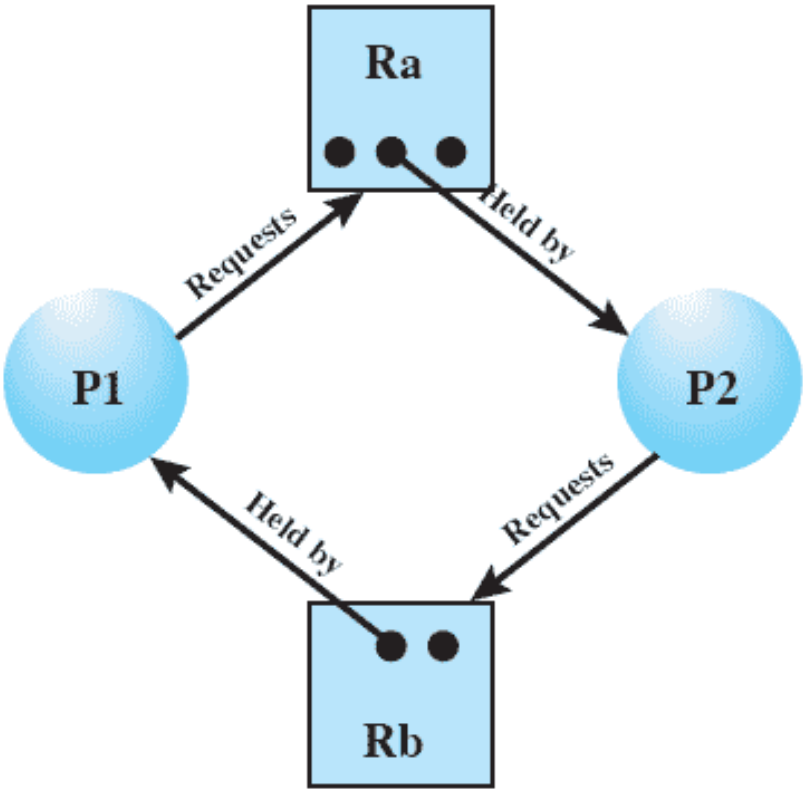
---

- No preemption
  - Tidak ada sumber daya yang dapat dipaksa dihilangkan dari suatu proses yang memegangnya
- Circular wait
  - Adanya rantai tertutup (*closed-chain*) proses-proses, sehingga setiap proses memegang setidaknya satu sumber daya yang diperlukan oleh proses berikutnya dalam rantai tersebut

# Contoh Grafik Alokasi Resource



(c) Circular wait



(d) No deadlock

# Contoh Grafik Alokasi Resource

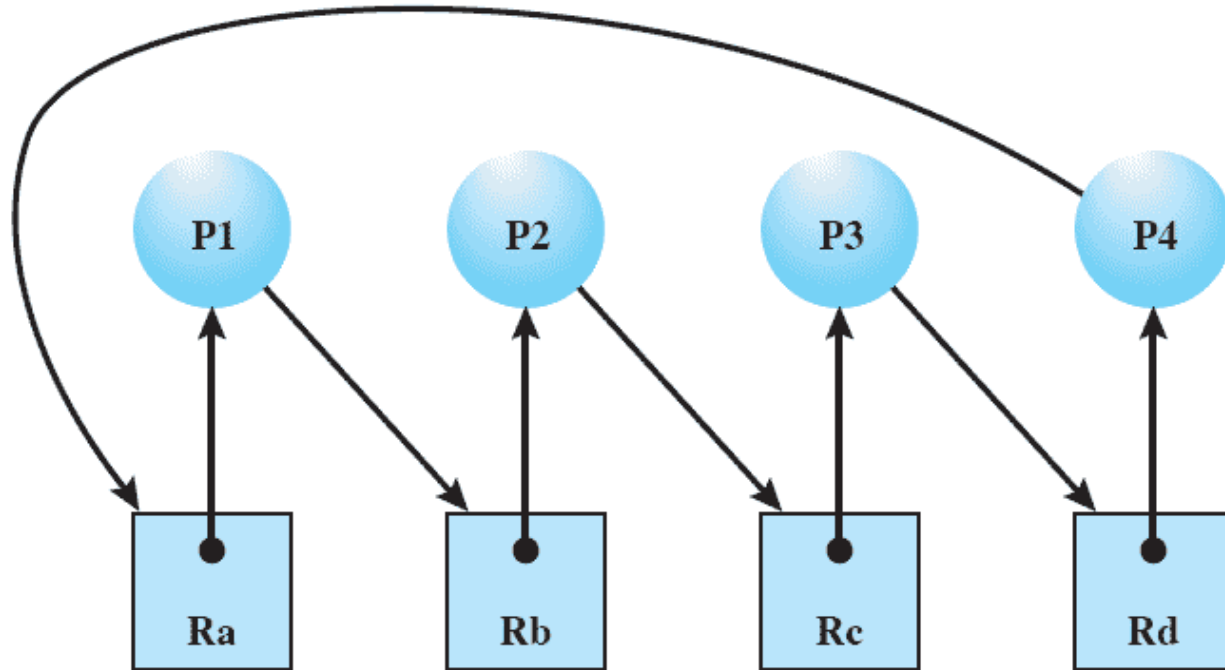


Figure 6.6 Resource Allocation Graph for Figure 6.1b

# Kemungkinan Deadlock

---

- Mutual Exclusion
- No preemption
- Hold and wait

# Terwujudnya Deadlock

---

- Mutual Exclusion
- No preemption
- Hold and wait
- Circular wait

# Pencegahan Deadlock (1)

---

- Mutual Exclusion
  - Harus didukung oleh SO
- Hold and Wait
  - Mengharuskan suatu proses meminta (*request*) semua sumber daya yang dibutuhkannya pada satu waktu

# Pencegahan Deadlock (2)

---

- No Preemption
  - Proses harus melepaskan sumber daya dan *request* lagi
  - SO boleh *preempt* suatu proses untuk mengharuskannya melepas sumber dayanya
- Circular Wait
  - Definisikan suatu pengurutan linier dari jenis-jenis sumber daya

# Penghindaran (*Avoidance*) Deadlock

---

- Suatu keputusan dibuat secara dinamis apakah permintaan alokasi sumber daya terkini akan, jika diijinkan, secara potential mengarah ke deadlock
- Memerlukan pengetahuan permintaan proses mendatang

# Pendekatan *Deadlock Avoidance*

---

- Jangan mulai suatu proses jikauntutannya dapat mengarah ke deadlock
- Jangan ijinikan suatu permintaan sumber daya berikutnya (*incremental*) untuk suatu proses jika alokasi ini dapat mengarah ke deadlock

# Penolakan Alokasi *Resource*

---

- Dikenal sebagai algoritma banker
- Status dari sistem adalah alokasi terkini dari sumber daya kepada proses
- Status aman (***safe***) adalah dimana terdapat setidaknya satu deretan yang tidak menghasilkan deadlock
- Status unsafe adalah status yang tidak aman

# Penentuan Status Safe (1)

	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
P1	9	3	6

Resource vector **R**

	R1	R2	R3
	0	1	1

Available vector **V**

(a) Initial state

# Penentuan Status Safe (2)

	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

# Penentuan Status Safe (3)

	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

# Penentuan Status Safe (4)

	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

# Penentuan Status Unsafe

	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

# Logika *Deadlock Avoidance* (1)

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

(a) global data structures

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

(b) resource alloc algorithm

# Logika *Deadlock Avoidance* (2)

```
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);
}
```

(c) test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic

# *Deadlock Avoidance*

---

- Kebutuhan *resource* maksimum harus dinyatakan sebelumnya
- Proses di bawah konsiderasi harus bersifat independen; tidak ada kebutuhan sinkronisasi
- Harus ada sejumlah fix sumber daya yang akan dialokasikan
- Tidak ada proses yang boleh ***exit*** selama memegang resource

# Deteksi Deadlock

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

	R1	R2	R3	R4	R5
2	1	1	2	1	

Resource vector

	R1	R2	R3	R4	R5
0	0	0	0	0	1

Allocation vector

**Figure 6.10 Example for Deadlock Detection**

# Strategi Saat Deadlock Terdeteksi

---

- Batalkan semua proses *terdeadlock*
- *Back up* setiap proses *terdeadlock* ke beberapa *checkpoint* terdefinisi sebelumnya & *restart* semua proses
  - Mungkin terjadi *original deadlock*

# Strategi Saat Deadlock Terdeteksi

---

- Berikutnya batalkan proses terdeadlock sampai *deadlock* tidak ada
- Kemudian *preempt* sumber daya sampai *deadlock* hilang (habis)

# Keuntungan - Kerugian

**Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]**

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	<ul style="list-style-type: none"> <li>• Works well for processes that perform a single burst of activity</li> <li>• No preemption necessary</li> </ul>	<ul style="list-style-type: none"> <li>• Inefficient</li> <li>• Delays process initiation</li> <li>• Future resource requirements must be known by processes</li> </ul>
		Preemption	<ul style="list-style-type: none"> <li>• Convenient when applied to resources whose state can be saved and restored easily</li> </ul>	<ul style="list-style-type: none"> <li>• Preempts more often than necessary</li> </ul>
		Resource ordering	<ul style="list-style-type: none"> <li>• Feasible to enforce via compile-time checks</li> <li>• Needs no run-time computation since problem is solved in system design</li> </ul>	<ul style="list-style-type: none"> <li>• Disallows incremental resource requests</li> </ul>
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	<ul style="list-style-type: none"> <li>• No preemption necessary</li> </ul>	<ul style="list-style-type: none"> <li>• Future resource requirements must be known by OS</li> <li>• Processes can be blocked for long periods</li> </ul>
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	<ul style="list-style-type: none"> <li>• Never delays process initiation</li> <li>• Facilitates online handling</li> </ul>	<ul style="list-style-type: none"> <li>• Inherent preemption losses</li> </ul>

# Masalah *Dining Philosophers* (1)

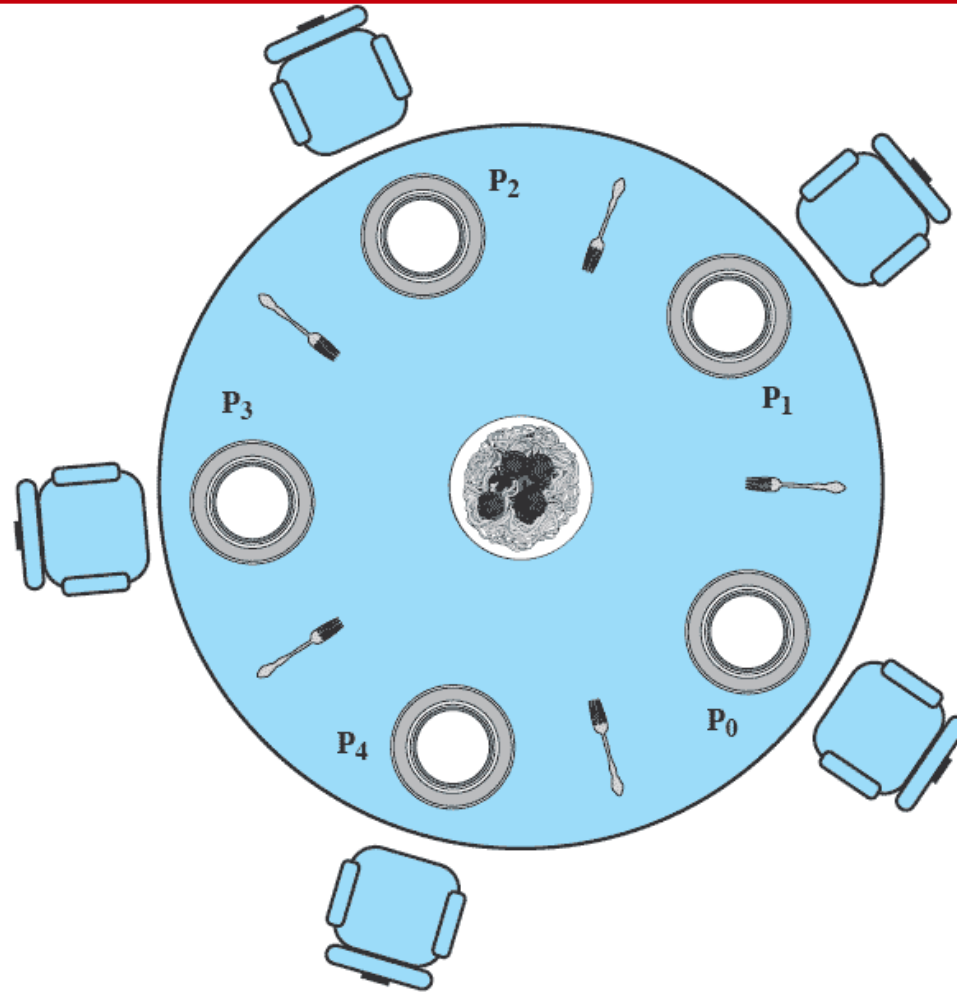


Figure 6.11 Dining Arrangement for Philosophers

# Masalah *Dining Philosophers* (2)

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
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()
{
    parbegin (philosopher (0), philosopher (1), philosopher
(2),
             philosopher (3), philosopher (4));
}
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

# Masalah *Dining Philosophers* (3)

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
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()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
             philosopher (3), philosopher (4));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

# Masalah *Dining Philosophers* (4)

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};   /* availability status of each fork */

void get_forks(int pid)     /* pid is the philosopher id number */
{
    int left = pid;
    int right = (++pid) % 5;
    /*grant the left fork*/
    if (!fork(left))
        cwait(ForkReady[left]);          /* queue on condition variable */
    fork(left) = false;
    /*grant the right fork*/
    if (!fork(right))
        cwait(ForkReady[right]);        /* queue on condition variable */
    fork(right) = false;
}
void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /*release the left fork*/
    if (empty(ForkReady[left])          /*no one is waiting for this fork */
        fork(left) = true;
    else                                /* awaken a process waiting on this fork */
        csignal(ForkReady[left]);
    /*release the right fork*/
    if (empty(ForkReady[right])         /*no one is waiting for this fork */
        fork(right) = true;
    else                                /* awaken a process waiting on this fork */
        csignal(ForkReady[right]);
}
```

# Masalah *Dining Philosophers* (5)

---

```
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 */
    }
}
```

**Figure 6.14** A Solution to the Dining Philosophers Problem Using a Monitor

# Mekanisme *Concurrency* UNIX

---

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals

# Sinyal UNIX

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

# Mekanisme *Concurrency* Linux

---

- Menyertakan semua mekanisme yang terdapat pada UNIX
- Operasi *atomic* berjalan tanpa interupsi dan tanpa interferensi

# Operasi *Atomic* Linux (1)

Table 6.3 Linux Atomic Operations

Atomic Integer Operations	
ATOMIC_INIT (int i)	At declaration: initialize an atomic_t to i
int atomic_read(atomic_t *v)	Read integer value of v
void atomic_set(atomic_t *v, int i)	Set the value of v to integer i
void atomic_add(int i, atomic_t *v)	Add i to v
void atomic_sub(int i, atomic_t *v)	Subtract i from v
void atomic_inc(atomic_t *v)	Add 1 to v
void atomic_dec(atomic_t *v)	Subtract 1 from v
int atomic_sub_and_test(int i, atomic_t *v)	Subtract i from v; return 1 if the result is zero; return 0 otherwise
int atomic_add_negative(int i, atomic_t *v)	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
int atomic_dec_and_test(atomic_t *v)	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
int atomic_inc_and_test(atomic_t *v)	Add 1 to v; return 1 if the result is zero; return 0 otherwise

# Operasi *Atomic* Linux (1)

Atomic Bitmap Operations	
<code>void set_bit(int nr, void *addr)</code>	Set bit nr in the bitmap pointed to by addr
<code>void clear_bit(int nr, void *addr)</code>	Clear bit nr in the bitmap pointed to by addr
<code>void change_bit(int nr, void *addr)</code>	Invert bit nr in the bitmap pointed to by addr
<code>int test_and_set_bit(int nr, void *addr)</code>	Set bit nr in the bitmap pointed to by addr; return the old bit value
<code>int test_and_clear_bit(int nr, void *addr)</code>	Clear bit nr in the bitmap pointed to by addr; return the old bit value
<code>int test_and_change_bit(int nr, void *addr)</code>	Invert bit nr in the bitmap pointed to by addr; return the old bit value
<code>int test_bit(int nr, void *addr)</code>	Return the value of bit nr in the bitmap pointed to by addr

# Spinlock Linux

<code>void spin_lock(spinlock_t *lock)</code>	Acquires the specified lock, spinning if needed until it is available
<code>void spin_lock_irq(spinlock_t *lock)</code>	Like <code>spin_lock</code> , but also disables interrupts on the local processor
<code>void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)</code>	Like <code>spin_lock_irq</code> , but also saves the current interrupt state in <code>flags</code>
<code>void spin_lock_bh(spinlock_t *lock)</code>	Like <code>spin_lock</code> , but also disables the execution of all bottom halves
<code>void spin_unlock(spinlock_t *lock)</code>	Releases given lock
<code>void spin_unlock_irq(spinlock_t *lock)</code>	Releases given lock and enables local interrupts
<code>void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)</code>	Releases given lock and restores local interrupts to given previous state
<code>void spin_unlock_bh(spinlock_t *lock)</code>	Releases given lock and enables bottom halves
<code>void spin_lock_init(spinlock_t *lock)</code>	Initializes given spinlock
<code>int spin_trylock(spinlock_t *lock)</code>	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
<code>int spin_is_locked(spinlock_t *lock)</code>	Returns nonzero if lock is currently held and zero otherwise

# Semaphore di Linux

Traditional Semaphores	
<code>void sema_init(struct semaphore *sem, int count)</code>	Initializes the dynamically created semaphore to the given count
<code>void init_MUTEX(struct semaphore *sem)</code>	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
<code>void init_MUTEX_LOCKED(struct semaphore *sem)</code>	Initializes the dynamically created semaphore with a count of 0 (initially locked)
<code>void down(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
<code>int down_interruptible(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received.
<code>int down_trylock(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
<code>void up(struct semaphore *sem)</code>	Releases the given semaphore
Reader-Writer Semaphores	
<code>void init_rwsem(struct rw_semaphore, *rwsem)</code>	Initializes the dynamically created semaphore with a count of 1
<code>void down_read(struct rw_semaphore, *rwsem)</code>	Down operation for readers
<code>void up_read(struct rw_semaphore, *rwsem)</code>	Up operation for readers
<code>void down_write(struct rw_semaphore, *rwsem)</code>	Down operation for writers
<code>void up_write(struct rw_semaphore, *rwsem)</code>	Up operation for writers

# Operasi Barrier Memory Linux

**Table 6.6 Linux Memory Barrier Operations**

<code>rmb()</code>	Prevents loads from being reordered across the barrier
<code>wmb()</code>	Prevents stores from being reordered across the barrier
<code>mb()</code>	Prevents loads and stores from being reordered across the barrier
<code>Barrier()</code>	Prevents the compiler from reordering loads or stores across the barrier
<code>smp_rmb()</code>	On SMP, provides a <code>rmb()</code> and on UP provides a <code>barrier()</code>
<code>smp_wmb()</code>	On SMP, provides a <code>wmb()</code> and on UP provides a <code>barrier()</code>
<code>smp_mb()</code>	On SMP, provides a <code>mb()</code> and on UP provides a <code>barrier()</code>

SMP = symmetric multiprocessor

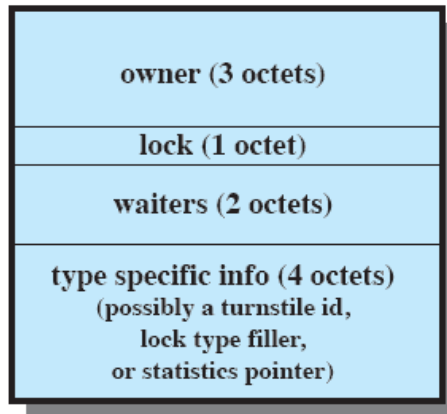
UP = uniprocessor

# Primitif Sinkronisasi Thread Solaris

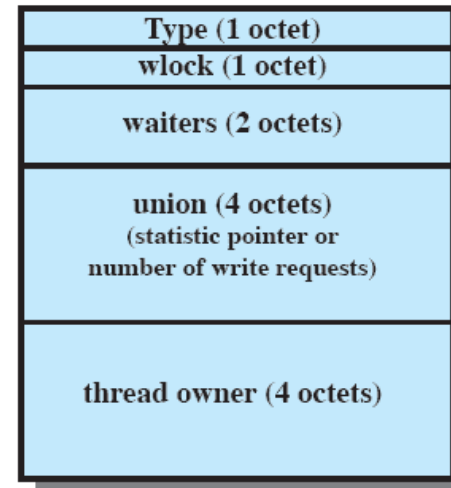
---

- Kunci mutual exclusion (mutex)
- Semaphores
- Kunci banyak reader, satu writer (readers/writer)
- Variabel kondisi

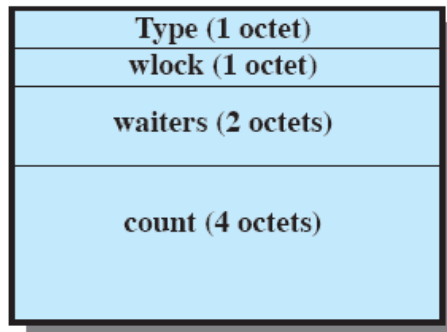
# Struktur Data Sinkronisasi Solaris



(a) MUTEX lock



(c) Reader/writer lock



(b) Semaphore



(d) Condition variable

Figure 6.15 Solaris Synchronization Data Structures

# Obyek Sinkronisasi Windows

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Notification Event	An announcement that a system event has occurred	Thread sets the event	All released
Synchronization event	An announcement that a system event has occurred.	Thread sets the event	One thread released
Mutex	A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore	Owning thread or other thread releases the mutex	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Waitable timer	A counter that records the passage of time	Set time arrives or time interval expires	All released
File	An instance of an opened file or I/O device	I/O operation completes	All released
Process	A program invocation, including the address space and resources required to run the program	Last thread terminates	All released
Thread	An executable entity within a process	Thread terminates	All released

*Note:* Shaded rows correspond to objects that exist for the sole purpose of synchronization.

# Tugas Pertemuan 6

---

- Jelaskan mekanisme penanganan **deadlock** di Windows dan Linux!
- Uraikan penyelesaian masalah Dining Philosophers Problem!
- Kerjakan problems 6.1 & 6.2!