Operating Systems

"Deadlock and Starvation"

Learning Objectives

- List and explain the conditions for deadlock.
- Define deadlock prevention and its strategies related to each of the conditions for deadlock.
- Explain the difference between deadlock prevention and deadlock avoidance.
- Understand two approaches to deadlock avoidance.
- Explain the fundamental difference in approach between deadlock detection and deadlock prevention or avoidance.
- Understand how an integrated deadlock strategy can be designed.
- Analyze the dining philosopher's problem.
- Explain the concurrency and synchronization methods used in UNIX, Linux, Solaris, and Windows 7.

Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution
- Involve conflicting needs for resources by two or more processes

Deadlock (illustration)

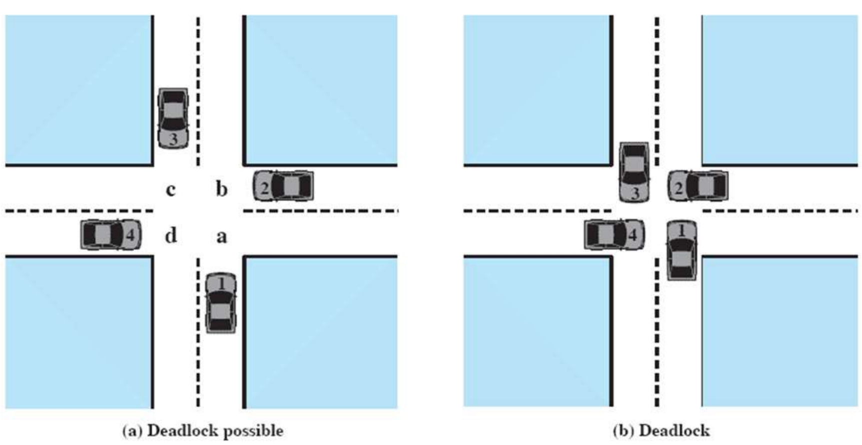


Figure 6.1 Illustration of Deadlock

Example of Deadlock

Two processes, P and Q, have the following general form:

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Get B	Get A
• • •	• • •
Release A	Release B
• • •	• • •
Release B	Release A
• • •	• • •

Deadlock

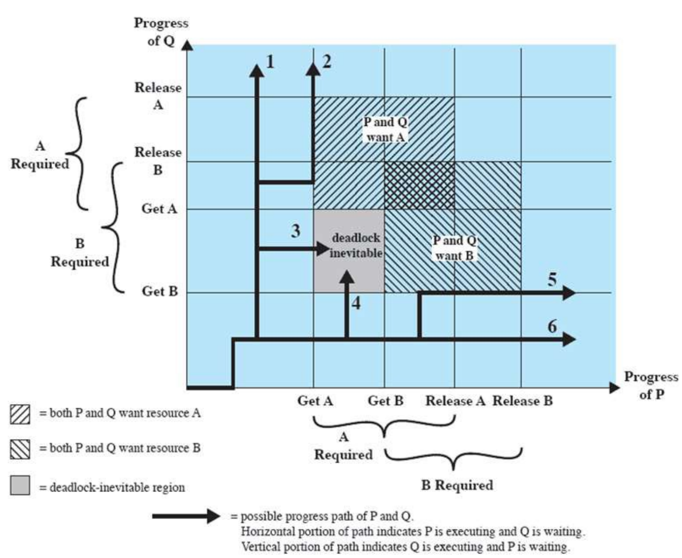


Figure 6.2 Example of Deadlock

Example of No Deadlock

- Changing the dynamics of the execution

Process P	**Process Q**
• • •	• • •
Get A	Get B
• • •	• • •
Release A	Get A
• • •	• • •
Get B	Release B
• • •	• • •
Release B	Release A
• • •	• • •

Deadlock

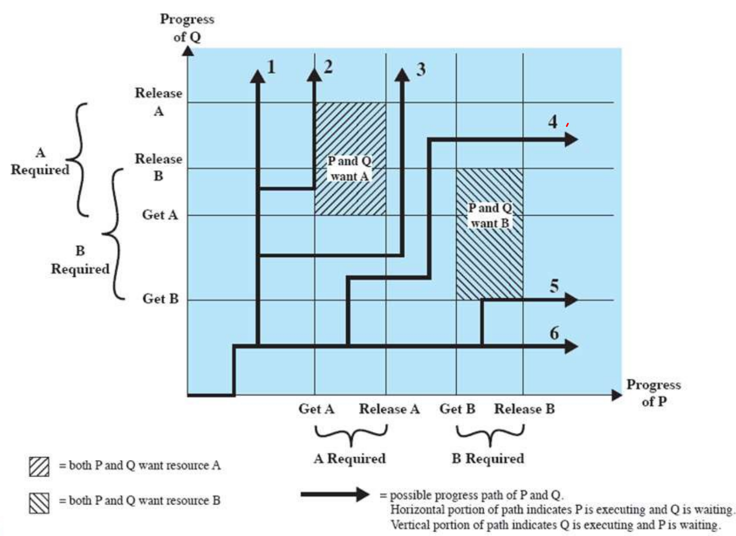


Figure 6.3 Example of No Deadlock [BACO03]

Reusable Resources

- Used by only one process at a time and not depleted by that use
- Processes obtain resources that they later release for reuse by other processes
- Example of reusable resources:
  - Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other

Reusable Resources

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

Reusable Resources

- Space is available for allocation of 200Kbytes, and the following sequence of events occur

P1	P2
. . .	. . .
Request 80 Kbytes;	Request 70 Kbytes;
. . .	. . .
Request 60 Kbytes;	Request 80 Kbytes;

- Deadlock occurs if both processes progress to their second request

Consumable Resources

- Created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

Example of Deadlock

- Deadlock occurs if receives blocking

```
P1
. . .
Receive(P2);
. . .
Send(P2, M1);
```

```
P2
. . .
Receive(P1);
. . .
Send(P1, M2);
```

Resource Allocation Graphs

- Directed graph that depicts a state of the system of resources and processes

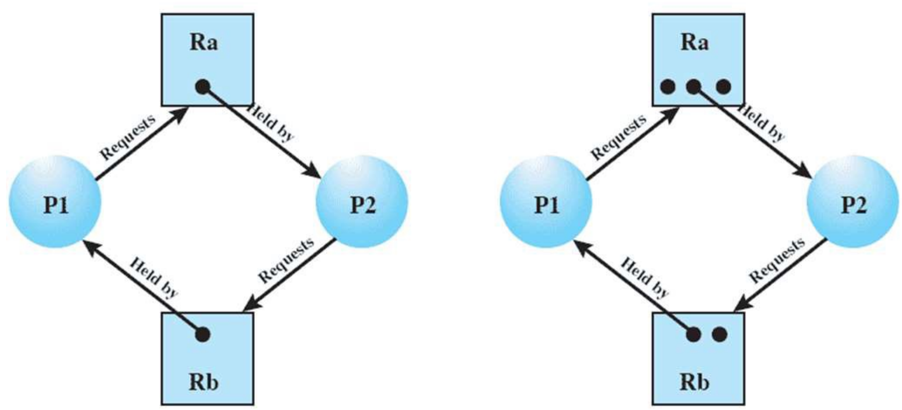
(a) Resouce is requested

(b) Resource is held

Conditions for Deadlock

- Mutual exclusion
  - Only one process may use a resource at a time
- Hold-and-wait
  - A process may hold allocated resources while awaiting assignment of others
- No preemption
  - No resource can be forcibly removed form 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

Resource Allocation Graphs



(c) Circular wait

(d) No deadlock

Resource Allocation Graphs

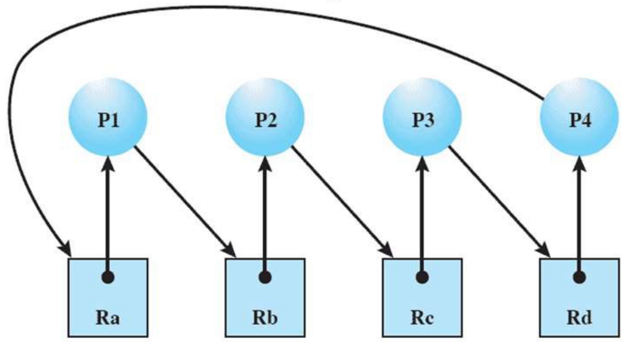
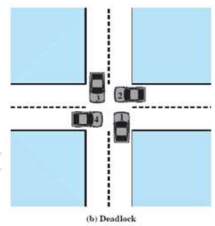


Figure 6.6 Resource Allocation Graph for



To summarize

Possibility of Deadlock	Existence of Deadlock
1. Mutual exclusion	**1.** Mutual exclusion
2. No preemption	**2.** No preemption
3. Hold and wait	**3.** Hold and wait
	4. Circular wait

Methods

- **Deadlock prevention**: Disallow one of the three necessary conditions (possible deadlock) for deadlock occurrence or prevent circular wait condition from happening.
- **Deadlock avoidance**: Do not grant a resource request if this allocation might lead to deadlock.
- **Deadlock detection**: Grant resource requests, when possible, but periodically check for the presence of deadlock and take action to recover.

Deadlock Prevention

- Mutual Exclusion
  - Must be supported by the OS and cannot be disallowed
- Hold and Wait
  - Require a process request all of its required resources at one time
  - To give required resources of a process at the same time
- No Preemption
  - Process must release resource and request again
  - OS may preempt a process to require it releases its resources
- Circular Wait
  - Define a linear ordering of resource types

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:
  - Do not start a process if its demands might lead to deadlock
  - Do not grant an incremental resource request to a process if this allocation might lead to deadlock

Resource Allocation Denial

- Referred to as **the banker's algorithm**
- State of the system is the current allocation of resources to process
- Safe state is where there is at least one sequence that does not result in deadlock
- Unsafe state is a state that is not safe

Determination of a Safe State

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
0	1	1

(a) Initial state

Determination of a Safe State

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
6	2	3

(b) P2 runs to completion

Determination of a Safe State

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
7	2	3

(c) P1 runs to completion

Determination of a Safe State

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
9	3	4

(d) P3 runs to completion

Determination of an Unsafe State (practice)

	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

- Deadlock avoidance has the advantage that it is not necessary to preempt and rollback processes, as in deadlock detection, and is less restrictive than deadlock prevention. However, it does have a number of restrictions on its use:
  - Maximum resource requirement must be stated in advance
  - Processes under consideration must be independent; no synchronization requirements
  - There must be a fixed number of resources to allocate
  - No process may exit while holding resources

Deadlock Avoidance Logic

```
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

Deadlock Avoidance Logic

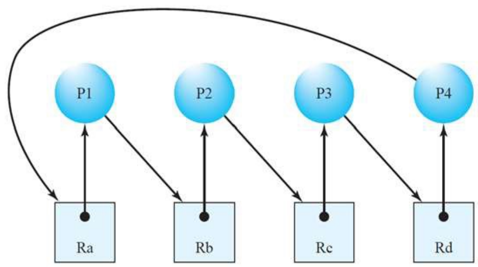
```
boolean safe (state S) {
   int currentavail[m];
   process rest[<number of processes>];
   currentavail = available;
   rest = {all processes};
   possible = true;
   while (possible) {
      <find a process Pk in rest such that
         claim [k,*] – alloc [k,*] <= currentavail;>
      if (found)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

- Requested resources are granted to processes whenever possible.
- Periodically, the OS performs an algorithm that allows it to detect the circular wait condition described earlier in condition



Deadlock Detection Alg

- A request matrix *Q* is defined such that *Qij* represents the amount of resources of type *j* requested by process *i*. The algorithm proceeds by marking processes that are not deadlocked. Initially, all processes are unmarked. Then the following steps are performed:
  - Mark each process that has a row in the Allocation matrix of all zeros. A process that has no allocated resources cannot participate in a deadlock.
  - Initialize a temporary vector **W** to equal the Available vector
  - Find an index i such that process i is currently unmarked and the ith row of **Q** is less than or equal to **W**. That is, Qik <= Wk, for 1 <= k <= m. If no such row is found, terminate the algorithm → **deadlock occurs**
  - If such a row is found, mark process i and add the corresponding row of the allocation matrix to W. That is, set $Wk = Wk + Aik$, for 1 <= k <= m. Return to step 3.

Deadlock Detection

	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	1

Allocation vector

Figure 6.10 Example for Deadlock Detection

Strategies Once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may occur
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Advantages and Disadvantages

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

Dining Philosophers Problem

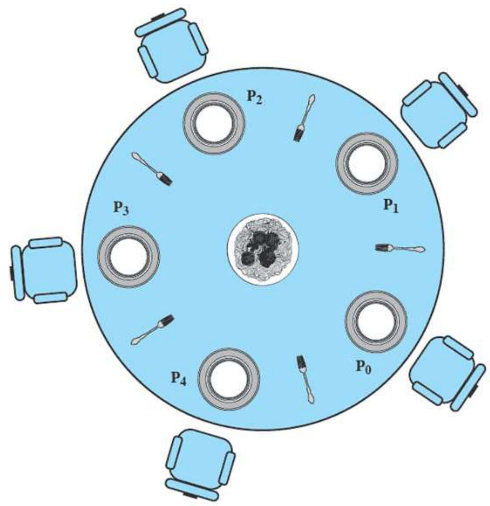


Figure 6.11 Dining Arrangement for Philosophers

Dining Philosophers Problem

```
/* 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

Dining Philosophers Problem

```
/* 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

Dining Philosophers Problem

```
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]);
}
```

Dining Philosophers Problem

```
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

UNIX Signals

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

UNIX Concurrency Mechanisms

- Pipes
- Messages
- Shared memory
- Semaphores
- Signals

Linux Kernel Concurrency Mechanism

- Includes all the mechanisms found in UNIX
- Atomic operations execute without interruption and without interference

Linux Atomic Operations

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

Linux Atomic Operations

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

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)	Like spin_lock, but also disables interrupts on the local processor
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)	Like spin_lock_irq, but also saves the current interrupt state in flags
void spin_lock_bh(spinlock_t *lock)	Like spin_lock, but also disables the execution of all bottom halves
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_unlock_bh(spinlock_t *lock)	Releases given lock and enables bottom halves
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)	Initializes the dynamically created semaphore to the given count
void init_MUTEX(struct semaphore *sem)	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
void init_MUTEX_LOCKED(struct semaphore *sem)	Initializes the dynamically created semaphore with a count of 0 (initially locked)
void down(struct semaphore *sem)	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
int down_interruptible(struct semaphore *sem)	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.
int down_trylock(struct semaphore *sem)	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
void up(struct semaphore *sem)	Releases the given semaphore
Reader-Writer Semaphores	
void init_rwsem(struct rw_semaphore, *rwsem)	Initalizes the dynamically created semaphore with a count of 1
void down_read(struct rw_semaphore, *rwsem)	Down operation for readers
void up_read(struct rw_semaphore, *rwsem)	Up operation for readers
void down_write(struct rw_semaphore, *rwsem)	Down operation for writers
void up_write(struct rw_semaphore, *rwsem)	Up operation for writers

Linux Memory Barrier Operations

Table 6.6 Linux Memory Barrier Operations

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

SMP = symmetric multiprocessor
UP = uniprocessor

Solaris Thread Synchronization Primitives

- Mutual exclusion (mutex) locks
- Semaphores
- Multiple readers, single writer (readers/writer) locks
- Condition variables

Solaris Synchronization Data Structures

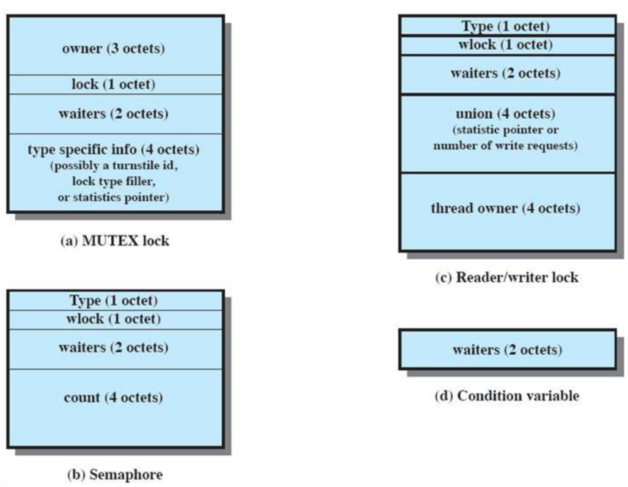


Figure 6.15 Solaris Synchronization Data Structures

Windows Synchronization Objects

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.

Thank You