CSC 553 Operating Systems

Lecture 6 - Concurrency: Deadlock and Starvation

Deadlock

- The ***permanent*** blocking of a set of processes that either compete for system 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 because none of the events is ever triggered
- No efficient solution in the general case

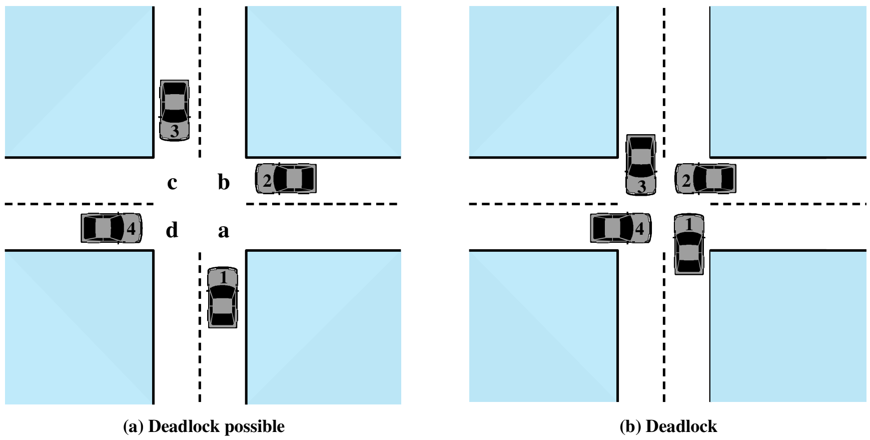

Figure 6.1 Illustration of Deadlock

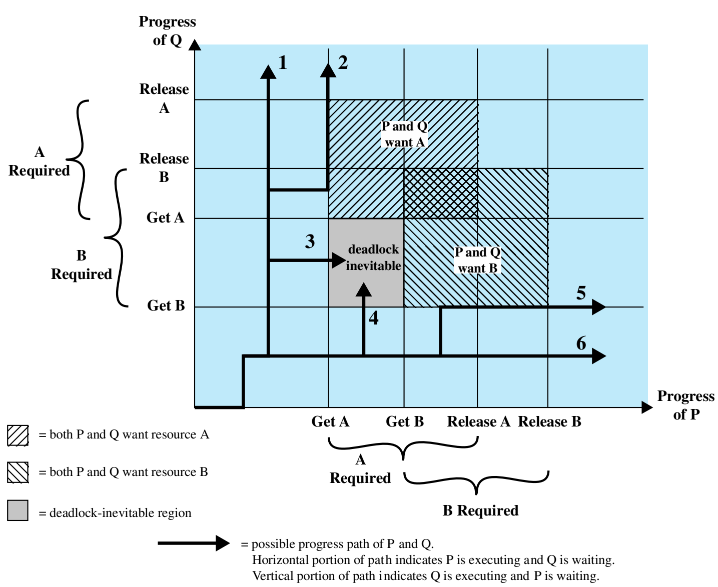

Figure 6.2 Example of Deadlock

The Six Execution Paths

1. **Q acquires B and then A and then releases B and A. When P resumes execution,** it will be able to acquire both resources.
2. **Q acquires B and then A. P executes and blocks on a request for A. Q releases** B and A. When P resumes execution, it will be able to acquire both resources.
3. **Q acquires B and then P acquires A. Deadlock is inevitable, because as execution** proceeds, Q will block on A and P will block on B.

The Six Execution Paths

4. **P acquires A and then Q acquires B. Deadlock is inevitable, because as execution** proceeds, Q will block on A and P will block on B.
5. **P acquires A and then B. Q executes and blocks on a request for B. P releases** A and B. When Q resumes execution, it will be able to acquire both resources.
6. **P acquires A and then B and then releases A and B. When Q resumes execution,** it will be able to acquire both resources.

Resource Categories

Reusable

- Can be safely used by only one process at a time and is not depleted by that use
  - Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores

Consumable

- One that can be created (produced) and destroyed (consumed)
  - Interrupts, signals, messages, and information
  - In I/O buffers

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

Deadlock occurs if the multiprogramming system interleaves the execution of the two processes as follows:

p 0 p 1 q 0 q 1 p 2 q 2

Example - Memory Request

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

P1	
Request 80 Kbytes;	
Request 60 Kbytes;	

- Deadlock occurs if both processes progress to their second request

Consumable Resources Deadlock

- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:
- Deadlock occurs if the Receive is blocking (receiving process is blocked until the message is received)

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

No One Strategy - Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems

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

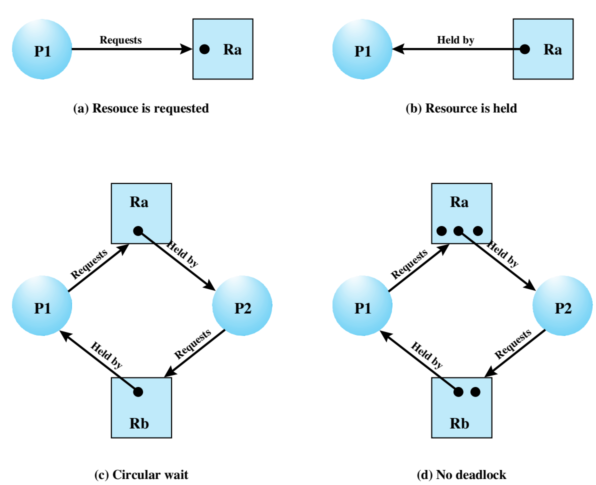

Figure 6.5 Examples of Resource Allocation Graphs

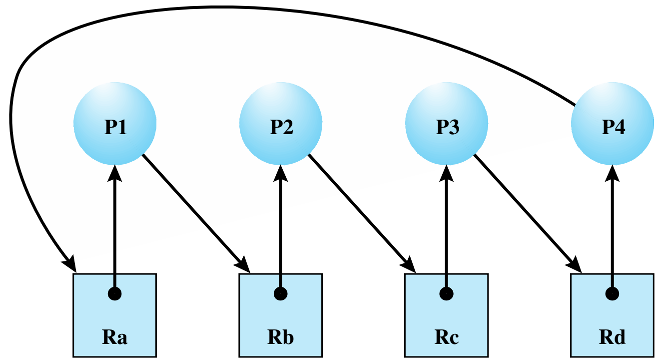

Figure 6.6 Resource Allocation Graph for Figure 6.1b

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 other resources.

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

- **Prevent Deadlock** - adopt a policy that eliminates one of the conditions
- **Avoid Dead**lock - make the appropriate dynamic choices based on the current state of resource allocation
- **Detect Deadl**ock - attempt to detect the presence of deadlock and take action to recover

Deadlock Prevention Strategy

- 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
  - **Direct** - prevent the occurrence of a circular wait

Deadlock Prevention

- **Mutual Exclusion** - if access to a resource requires mutual exclusion then it must be supported by the OS
- Hold and Wait
  - Require that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously
  - Inefficient because of the long wait for when they are all available and the long idle times for these resources

Deadlock Prevention

- No Preemption
  - if a process holding certain resources is denied a further request, that process must release its original resources and request them again
  - OS may preempt the second process and require it to release its resources (practical only when applied to resources whose state can be easily saved and restored later, as is the case with a processor)
- 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

Deadlock Avoidance

- Two 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

Resource Allocation Denial

- Referred to as 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
- ***Unsafe state*** is a state that is not safe

Determination of a Safe 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	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

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

P3 runs to completion

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	5	1	1
P3	2	1	1
P4	0	0	2

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
1	1	2

(a) Initial 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	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

C – A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
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

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

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

```
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)

Deadlock Avoidance Advantages

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

Deadlock Avoidance Restrictions

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

Deadlock Strategies

- Deadlock prevention strategies are very conservative
  - Limit access to resources by imposing restrictions on processes
- Deadlock detection strategies do the opposite
  - Resource requests are granted whenever possible

Deadlock Detection Algorithms

- A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur
- Advantages:
  - it leads to early detection
  - the algorithm is relatively simple
- Disadvantage
  - frequent checks consume considerable processor time

Deadlock Detection Algorithm

1. Mark P4, because P4 has no allocated resources.
2. Set **W** = (0 0 0 0 1).
3. The request of process P3 is less than or equal to **W**, so mark P3 and set W=W + (0 0 0 1 0) = (0 0 0 1 1).
4. No other unmarked process has a row in Q that is less than or equal to W.
   - Therefore, terminate the algorithm.
5. The algorithm concludes with P1 and P2 unmarked, indicating that these processes are deadlocked

	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

Recovery Strategies

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

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

Summary of Deadlock Detection, Prevention, and Avoidance Approaches

Dining Philosophers Problem

- No two philosophers can use the same fork at the same time (mutual exclusion)
- No philosopher must starve to death (avoid deadlock and starvation)

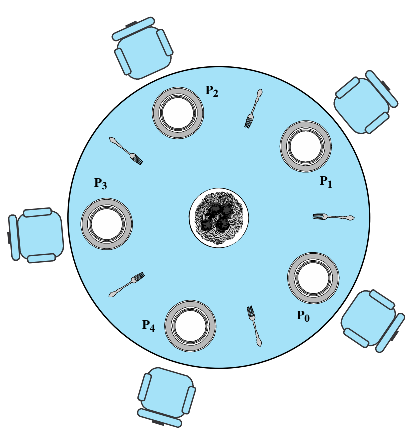

Figure 6.11 Dining Arrangement for Philosophers

Dining Philosophers Problem First Solution

```
/* 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));
     }
```

Leads to deadlock – all reach for a first fork and get it. They all wait for the second fork without yielding the first

Dining Philosophers Problem Second Solution

```
/* 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));
}
```

Dining Philosophers Solution Using a Monitor

```
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]);
}
```

```
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 */
   }
}
```

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:
  - Pipes
  - Messages
  - Shared Memory
  - Semaphores
  - Signals

UNIX Concurrency Mechanisms

- Pipes, messages, and shared memory can be used to communicate data between processes.
- Semaphores and signals are used to trigger actions by other processes.

Pipes

- Circular buffers allowing two processes to communicate on the producer-consumer model
  - First-in-first-out queue, written by one process and read by another
- Two types:
  - Named
  - Unnamed

Pipes

- When a pipe is created, it is given a fixed size in bytes.
- When a process attempts to write into the pipe, the write request is immediately executed if there is sufficient room; otherwise the process is blocked.
- A reading process is blocked if it attempts to read more bytes than are currently in the pipe; otherwise the read request is immediately executed.
- The OS enforces mutual exclusion: that is, only one process can access a pipe at a time.

Messages

- A block of bytes with an accompanying type
- UNIX provides ***msgsnd*** and ***msgrcv*** system calls for processes to engage in message passing
- Associated with each process is a message queue, which functions like a mailbox

Messages

- The message sender specifies the type of message with each message sent, and this can be used as a selection criterion by the receiver.
- The receiver can either retrieve messages in first-in-first-out order or by type.
- A process will block when trying to send a message to a full queue.
- A process will also block when trying to read from an empty queue. If a process attempts to read a message of a certain type and fails because no message of that type is present, the process is not blocked.

Shared Memory

- Fastest form of interprocess communication
- Common block of virtual memory shared by multiple processes
- Permission is read-only or read-write for a process
- Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory

Semaphores

- Generalization of the **`semWait`** and **`semSignal`** primitives
  - no other process may access the semaphore until all operations have completed

Semaphores

- Consists of:
  - current value of the semaphore
  - process ID of the last process to operate on the semaphore
  - number of processes waiting for the semaphore value to be greater than its current value
  - number of processes waiting for the semaphore value to be zero

Signals

- A software mechanism that informs a process of the occurrence of asynchronous events
  - similar to a hardware interrupt, but does not employ priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent
- A process may respond to a signal by:
  - performing some default action
  - executing a signal-handler function
  - ignoring the signal

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

Linux Kernel Concurrency Mechanism

- Includes all the mechanisms found in UNIX plus:
  - Spinlocks
  - Atomic Operations
  - Semaphores
  - Barriers

Atomic Operations

- Atomic operations execute without interruption and without interference
- Simplest of the approaches to kernel synchronization
- Two types:
  - Integer Operations
    - operate on an integer variable
    - typically used to implement counters
  - **Bitmap Operations** - operate on one of a sequence of bits at an arbitrary memory location indicated by a pointer variable

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
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 Atomic Operations

Spinlocks

- Most common technique for protecting a critical section in Linux
- Can only be acquired by one thread at a time; any other thread will keep trying (spinning) until it can acquire the lock
- Built on an integer location in memory that is checked by each thread before it enters its critical section

Spinlocks

- Effective in situations where the wait time for acquiring a lock is expected to be very short
- **<u>Disadvantage</u>**: locked-out threads continue to execute in a busy-waiting mode

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 Spin-locks

Semaphores

- User level:
  – Linux provides a semaphore interface corresponding to that in UNIX SVR4
- Internally:
  - Implemented as functions within the kernel and are more efficient than user-visible semaphores
- Three types of kernel semaphores:
  - Binary semaphores
  - Counting semaphores
  - Reader-writer semaphores

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)	Initializes 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

Barriers

- In parallel computing, a **barrier** is a type of synchronization method.
  - A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

Barriers

- To enforce the order in which instructions are executed, Linux provides the memory barrier facility.
  - **`rmb()`** - insures that no reads occur across the barrier defined by the place of the **`rmb()`** in the code.
  - **`wmb()`**– insures that no writes occur across the barrier defined by the place of the **`wmb()`** in the code.
  - **`mb()`** - provides both a load and store barrier.

Barriers

- Two important points:
  - The barriers relate to machine instructions, namely loads and stores.
  - The **`rmb`** , **`wmb`** , and **`mb`** operations dictate the behavior of both the compiler and the processor.

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

Synchronization Primitives

- In addition to the concurrency mechanisms of UNIX SVR4, Solaris supports four thread synchronization primitives:
  - Mutual exclusion (mutex) locks
  - Semaphores
  - Condition variables
  - Readers/writer locks

Synchronization Primitives

- Solaris implements these primitives within the kernel for kernel threads; they are also provided in the threads library for user-level threads.
- The initialization functions for the primitives fill in some of the data members.
- Once a synchronization object is created, there are essentially only two operations that can be performed:
  - enter (acquire lock)
  - release (unlock)

Synchronization Primitives

- There are no mechanisms in the kernel or the threads library to enforce mutual exclusion or to prevent deadlock.
- Threads can attempts to access "protected" data and code segments. if the appropriate synchronization primitive isn't used.
- If a thread locks an object and doesn't unlock it, no kernel action is taken.

Synchronization Primitives

- All of the synchronization primitives require the existence of a hardware instruction that allows an object to be tested and set in one atomic operation.

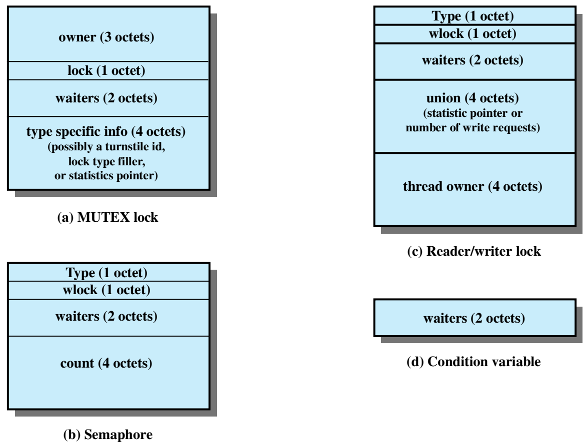

Figure 6.15 Solaris Synchronization Data Structures

Mutual Exclusion Lock

- Used to ensure only one thread at a time can access the resource protected by the mutex
- The thread that locks the mutex must be the one that unlocks it
- A thread attempts to acquire a mutex lock by executing the **`mutex_enter`** primitive
- Default blocking policy is a spinlock
- An interrupt-based blocking mechanism is optional

Semaphores

- Solaris provides classic counting semaphores with the following primitives:
  - **`sema_p()`** - decrements the semaphore, potentially blocking the thread
  - **`sema_v()`** - increments the semaphore, potentially unblocking a waiting thread
  - **`sema_tryp()`** - decrements the semaphore if blocking is not required

Readers/Writer Locks

- Allows multiple threads to have simultaneous read-only access to an object protected by the lock
- Allows a single thread to access the object for writing at one time, while excluding all readers
  - when lock is acquired for writing it takes on the status of **`write lock`**
  - if one or more readers have acquired the lock its status is **`read lock`**

Condition Variables

- A condition variable is used to wait until a particular condition is true
- Condition variables must be used in conjunction with a mutex lockThis implements a monitor of the type illustrated in the Dining Philosophers Problem

Summary

- Principles of deadlock
  - Reusable/consumable resources
  - Resource allocation graphs
  - Conditions for deadlock
- Deadlock prevention
    - Mutual exclusion
    - Hold and wait
    - No preemption
    - Circular wait

Summary

- Deadlock avoidance
    - Process initiation denial
    - Resource allocation denial
- Deadlock detection
    - Deadlock detection algorithm
    - Recovery
- UNIX concurrency mechanisms
    - Pipes
    - Messages
    - Shared memory
    - Semaphores
    - Signals

Summary

– Linux kernel concurrency mechanisms
    - Atomic operations
    - Spinlocks
    - Semaphores
    - Barriers
- Solaris thread synchronization primitives
    - Mutual exclusion lock
    - Semaphores
    - Readers/writer lock
    - Condition variables