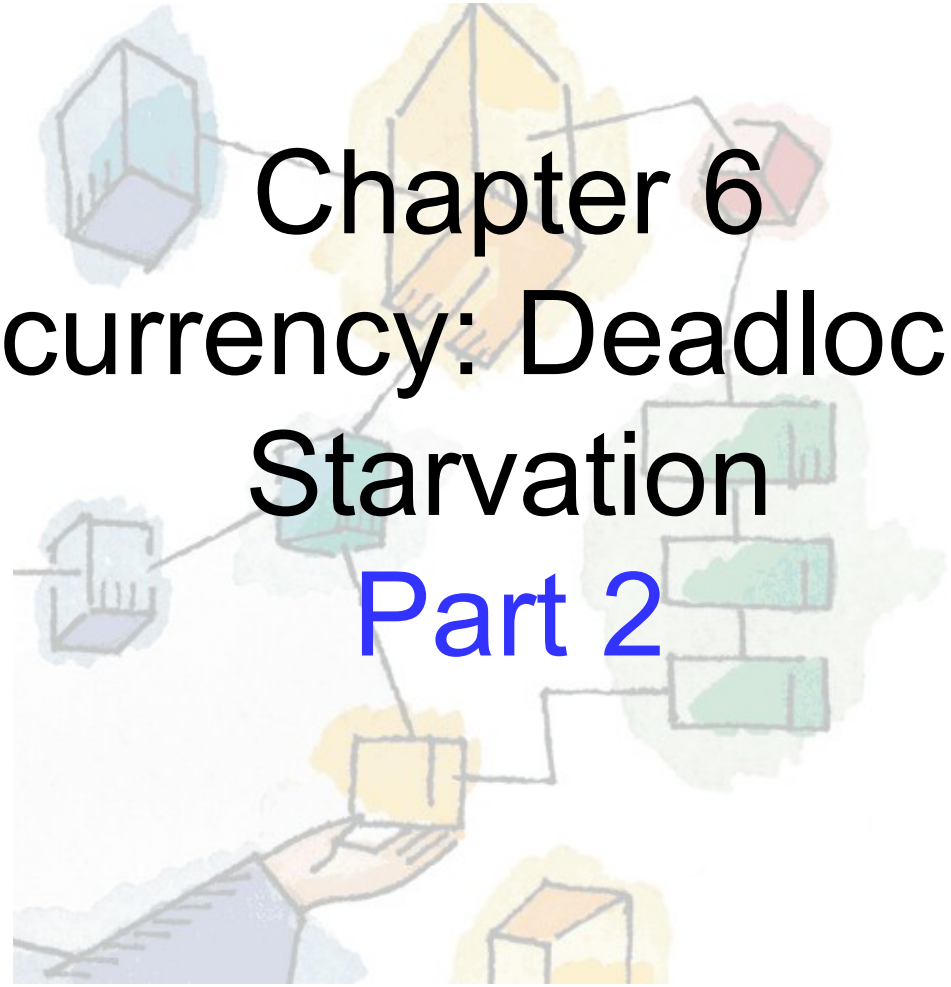
Operating Systems:
Internals and Design Principles, 8/E
William Stallings



Chapter 6
Concurrency: Deadlock and Starvation
Part 2

Patricia Roy
Manatee Community College, Venice, FL

Resource Allocation Denial

- The strategy referred to as the *banker's algorithm*
- State of the system is the current allocation of resources to processes:
  - Vectors: Resource & Available
  - Matrices: Claim & Allocation
- Safe state is where there is at least one sequence of resource allocation that does not result in deadlock
- Unsafe state is a state that is not safe

Deadlock Avoidance

- Differs subtly from deadlock prevention
- Allows the 3 necessary conditions, but make intelligent choices to assure that deadlock point is never reached
  - *Is the current resource alloc. will lead to potential deadlock?*
- Thus, it allows more concurrency than prevention
- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Thus, requires knowledge of future process requests

Determination of a Safe State

Can any of the 4 processes be run to completion with the resources available? (Cij – Aij <= Vij for all j)

What they request

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

Claim matrix **C**

What they get

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

Allocation matrix **A**

What they didn't get

	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

Total amount of resources

Resources available after allocation

Is this a safe state?

Cij – Aij <= Vij for all j

	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

P2 returns all of its resources

** Now, each of the remaining processes could be completed

Is this a safe state?

Suppose now P1 runs to completion, and returns all of its resources

	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

Is this a safe state?

Suppose now P3 runs to completion, and returns all of its resources

	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

*** Therefore, all of these processes are in the safe state

Determination of an Unsafe 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	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

Not possible, thus this is not a safe state!
Potential for DEADLOCK.
P1 request should be denied,
P1 should be blocked!

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 Detection

- Deadlock prevention strategies are very conservative
  - Solve deadlock by limiting access to resources
  - Imposing restrictions on processes

- Deadlock detection do not limit resource access
  - Requested resources are granted to processes whenever possible
  - The OS will periodically performs an algorithm to detect the circular wait condition

Deadlock Detection

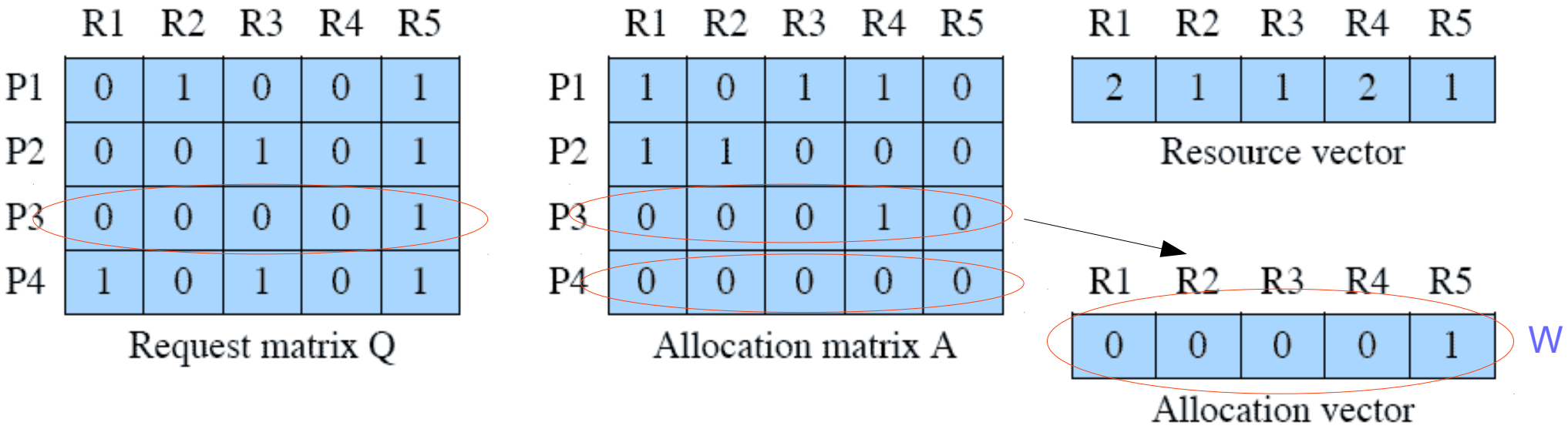
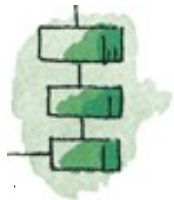


Figure 6.10 Example for Deadlock Detection

W = W + (00010) = (00011)

1. Mark each process that has a row in Allocation matrix of all zeros
2. Initialise a temporary vector W to equal the Available vector
3. Find an index *i* such that process *i* is currently unmarked and the *ith* row of Q is less than or equal to W. (Q*ik* <= W*k* for 1<=k<=m). If no such row is found, terminate the algorithm (DEADLOCK)
4. If such a row is found, mark process *i* and add the corresponding row of the allocation matrix to W. (W*k* = W*k* + A*ik* for 1<=k<=m). Return to step 3.



Strategies Once Deadlock Detected

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
  - Risk: Original deadlock may recur
- Successively abort deadlocked processes until deadlock no longer exists.
  - order depends on the minimum cost
- Successively preempt resources until deadlock no longer exists.
  - Also use minimum cost-based approach

Dining Philosophers Problem

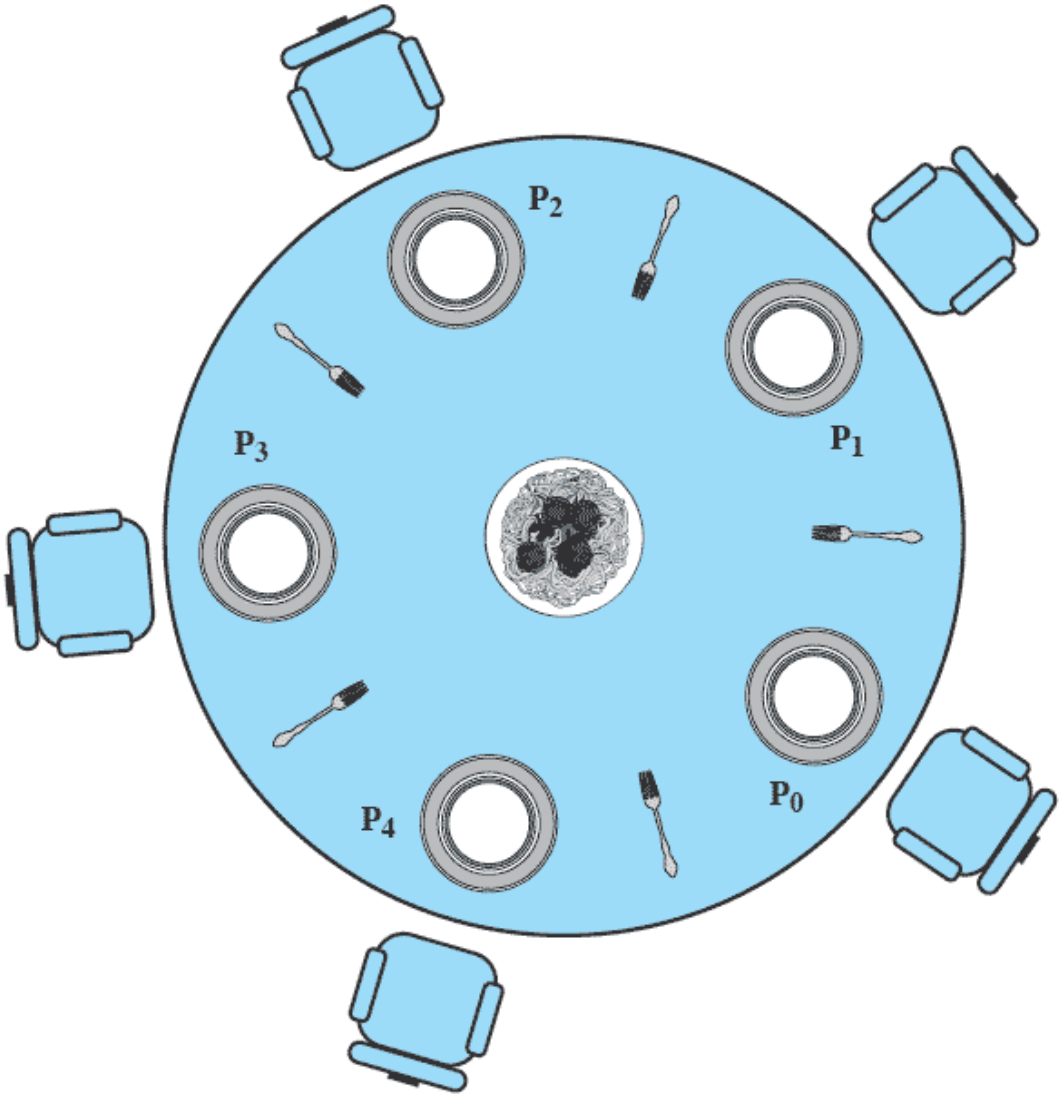


Figure 6.11 Dining Arrangement for Philosophers

5 philosophers in a house

Thinking & Eating!

The only food that contribute : Spaghetti!

2 forks each to eat spaghetti

Rule:

1. Satisfy mutual exclusion
   - No 2 philosophers can use the same forks at the same time
2. Avoid deadlock and starvation

Solutions...

```
/* 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

Solutions...

```
/* 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

UNIX Concurrency Mechanisms

- Pipes
  - Circular buffer allowing two processes to communicate on the producer-consumer model
- Messages
  - For processes to engage in msg passing – mailbox
  - Read from empty mailbox –> block
- Shared memory
  - Virtual memory shared by multiple processes
- Semaphores
  - SemWait & SemSignal
- Signals
  - Similar to hardware interrupt, but has no priorities

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 concurrency mechanisms found in UNIX (e.g: SVR4 including pipes, messages, shared memory, and signals)

- Three types:
  - Atomic operations
  - Spinlocks
  - Semaphores

Atomic operations

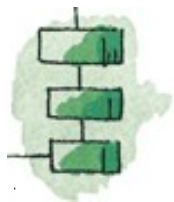
- Execute without interruption and without interference
- Use to avoid simple race conditions

- Simplest of the approaches to kernel synchronization
- Two Types:
  - Atomic integer operation
  - Atomic bitmap operation

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



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
  - If the value is 0, the thread enters the critical region and set the value to 1
  - If the value is nonzero, the thread will continually check the value until the value is 0.
- Effective in situations where the wait time for acquiring a lock is expected to be very short
- Disadvantage:
  - locked-out threads continue to execute in a busy-waiting mode

Spinlocks

```
spin_lock (&lock)
 /*critical section */
spin_unlock (&lock)
```

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

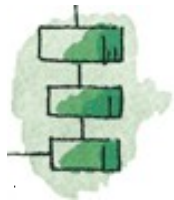
Semaphores

- 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
    - Multiple concurrent readers
    - Only one writer

Same as in Chapter 5

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



Summary

Deadlock:

- the blocking of a set of processes that either compete for system resources or communicate with each other
- blockage is permanent unless OS takes action
- may involve reusable or consumable resources
  - Consumable = destroyed when acquired by a process
  - Reusable = not depleted/destroyed by use
- **Dealing with deadlock:**
  - prevention – guarantees that deadlock will not occur
  - detection – OS checks for deadlock and takes action
  - avoidance – analyses each new resource request