

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"> •Works well for processes that perform a single burst of activity •No preemption necessary | <ul style="list-style-type: none"> •Inefficient •Delays process initiation •Future resource requirements must be known by processes |
| | | Preemption | <ul style="list-style-type: none"> •Convenient when applied to resources whose state can be saved and restored easily | <ul style="list-style-type: none"> •Preempts more often than necessary |
| | | Resource ordering | <ul style="list-style-type: none"> •Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design | <ul style="list-style-type: none"> •Disallows incremental resource requests |
| Avoidance | Midway between that of detection and prevention | Manipulate to find at least one safe path | <ul style="list-style-type: none"> •No preemption necessary | <ul style="list-style-type: none"> •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 | <ul style="list-style-type: none"> •Never delays process initiation •Facilitates online handling | <ul style="list-style-type: none"> •Inherent preemption losses |

Table 6.2 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 |

Table 6.3 Linux Atomic Operations

| Atomic Integer Operations | |
|--|--|
| <code>ATOMIC_INIT (int i)</code> | At declaration: initialize an <code>atomic_t</code> to <code>i</code> |
| <code>int atomic_read(atomic_t *v)</code> | Read integer value of <code>v</code> |
| <code>void atomic_set(atomic_t *v, int i)</code> | Set the value of <code>v</code> to integer <code>i</code> |
| <code>void atomic_add(int i, atomic_t *v)</code> | Add <code>i</code> to <code>v</code> |
| <code>void atomic_sub(int i, atomic_t *v)</code> | Subtract <code>i</code> from <code>v</code> |
| <code>void atomic_inc(atomic_t *v)</code> | Add 1 to <code>v</code> |
| <code>void atomic_dec(atomic_t *v)</code> | Subtract 1 from <code>v</code> |
| <code>int atomic_sub_and_test(int i, atomic_t *v)</code> | Subtract <code>i</code> from <code>v</code> ; return 1 if the result is zero; return 0 otherwise |
| <code>int atomic_add_negative(int i, atomic_t *v)</code> | Add <code>i</code> to <code>v</code> ; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores) |
| <code>int atomic_dec_and_test(atomic_t *v)</code> | Subtract 1 from <code>v</code> ; return 1 if the result is zero; return 0 otherwise |
| <code>int atomic_inc_and_test(atomic_t *v)</code> | Add 1 to <code>v</code> ; return 1 if the result is zero; return 0 otherwise |
| Atomic Bitmap Operations | |
| <code>void set_bit(int nr, void *addr)</code> | Set bit <code>nr</code> in the bitmap pointed to by <code>addr</code> |
| <code>void clear_bit(int nr, void *addr)</code> | Clear bit <code>nr</code> in the bitmap pointed to by <code>addr</code> |
| <code>void change_bit(int nr, void *addr)</code> | Invert bit <code>nr</code> in the bitmap pointed to by <code>addr</code> |
| <code>int test_and_set_bit(int nr, void *addr)</code> | Set bit <code>nr</code> in the bitmap pointed to by <code>addr</code> ; return the old bit value |
| <code>int test_and_clear_bit(int nr, void *addr)</code> | Clear bit <code>nr</code> in the bitmap pointed to by <code>addr</code> ; return the old bit value |
| <code>int test_and_change_bit(int nr, void *addr)</code> | Invert bit <code>nr</code> in the bitmap pointed to by <code>addr</code> ; return the old bit value |
| <code>int test_bit(int nr, void *addr)</code> | Return the value of bit <code>nr</code> in the bitmap pointed to by <code>addr</code> |

Table 6.4 Linux Spinlocks

| | |
|---|---|
| <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 flags |
| <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 |

Table 6.5 Linux Semaphores

| 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 |

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

Table 6.7 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.