'A Lightweight Semaphore for Linux'

Jojumon Kavalan   Joy Menon   Samveen Gulati

Department of Computer Science and Engineering, IIT Mumbai.

31 October 2004

What are Semaphores?

Definition: Semaphores are a mechanism to allow contending processes/ threads to query, alter, monitor and control access to shared system resources.

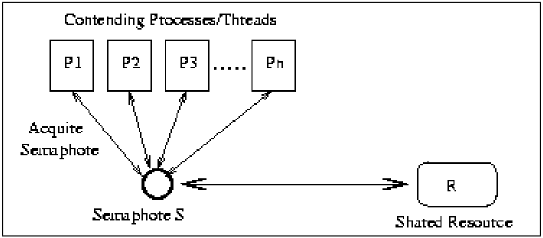


Figure: Semaphore S for shared resource R

More on Semaphores...

More on Semaphores...

- Types of Semaphores:
  - Binary Semaphores.
  - Counting Semaphores.
  - Semaphore Sets.

More on Semaphores...

- Types of Semaphores:
  - Binary Semaphores.
  - Counting Semaphores.
  - Semaphore Sets.

- Typical usage scenarios for Semaphores:
  1. Access to buffers in producer-consumer synchronization (binary semaphores)
  2. Access to shared memory segments. This can help serve as a mechanism for interprocess communication.
  3. Access to class members (static variables) by objects.

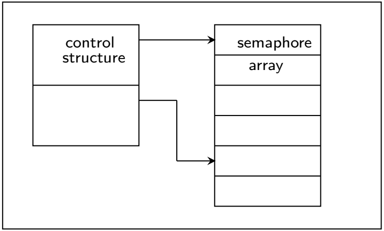
More on Semaphores...

- Types of Semaphores:
  - Binary Semaphores.
  - Counting Semaphores.
  - Semaphore Sets.
- Typical usage scenarios for Semaphores:
  1. Access to buffers in producer-consumer synchronization (binary semaphores)
  2. Access to shared memory segments. This can help serve as a mechanism for interprocess communication.
  3. Access to class members (static variables) by objects.
- Linux 2.4.x kernels use an implementation of the 'System V Semaphore' specification.

Set of Semaphores

Semaphore set is a control structure with a unique ID and an array of semaphores.

The identifier for the semaphore or array is called the semid.



Semaphore Performance

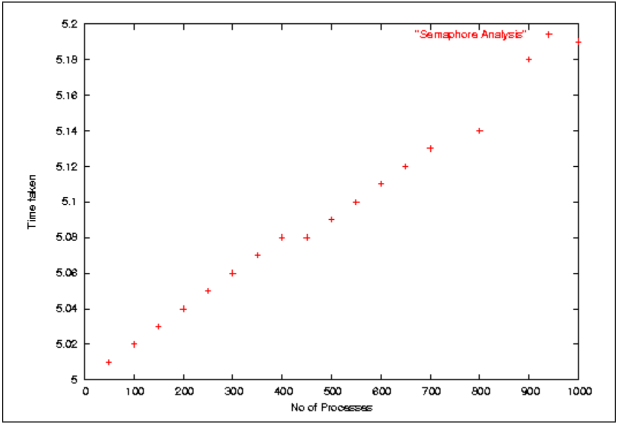


Figure: Graphical Analysis - Performance of current Linux Semaphores

Performance Issues

- The Linux semaphore implementation is 'heavyweight' in terms of memory-usage and performance.

Performance Issues

- The Linux semaphore implementation is 'heavyweight' in terms of memory-usage and performance.
- Reasons:
  - **Sets of semaphores** - size of sets may be a maximum of 25. Because of this usage of large 'sets', the implemetation is heavyweight.

Performance Issues

- The Linux semaphore implementation is 'heavyweight' in terms of memory-usage and performance.
- Reasons:
  - **Sets of semaphores** - size of sets may be a maximum of 25. Because of this usage of large 'sets', the implemetation is heavyweight.
  - **Heavy context switching** - typically 25-30% time while locking. See [3].

Performance Issues

- The Linux semaphore implementation is 'heavyweight' in terms of memory-usage and performance.
- Reasons:
  - **Sets of semaphores** - size of sets may be a maximum of 25. Because of this usage of large 'sets', the implemetation is heavyweight.
  - **Heavy context switching** - typically 25-30% time while locking. See [3].
- Linux needs a Lightweight Semaphore!

Design Approaches

- The following 3 design approaches were examined:

Design Approaches

- The following 3 design approaches were examined:
    1. Modify current algorithms to remove bottlenecks.

Design Approaches

- The following 3 design approaches were examined:

1. Modify current algorithms to remove bottlenecks.
2. Apply existing lightweight solutions.
   - Small-Memory Embedded Systems semaphores. See [3].

Design Approaches

- The following 3 design approaches were examined:

1. Modify current algorithms to remove bottlenecks.
2. Apply existing lightweight solutions.
   - Small-Memory Embedded Systems semaphores. See [3].
   - General Parallel File System semaphores. See [1] and [2].

Design Approaches

- The following 3 design approaches were examined:

1. Modify current algorithms to remove bottlenecks.
2. Apply existing lightweight solutions.
   - Small-Memory Embedded Systems semaphores. See [3].
   - General Parallel File System semaphores. See [1] and [2].
3. Use Single Semaphores instead of sets.

Single Semaphores Design

Single Semaphores Design

- Use a single semaphore instead of a set of semaphores.

Single Semaphores Design

- Use a single semaphore instead of a set of semaphores.
- Designed to overcome the limitations of set of semaphores.

Single Semaphores Design

- Use a single semaphore instead of a set of semaphores.
- Designed to overcome the limitations of set of semaphores.
- Effective utilisation of system resources.

Single Semaphores Design

- Use a single semaphore instead of a set of semaphores.
- Designed to overcome the limitations of set of semaphores.
- Effective utilisation of system resources.
- Enable the system to provide more semaphores for the application programs.

The Interface

The Interface Specification for the Implementation is as follows:

- **int s_semget (key_t key, int sem_flag, int sem_val);**
  This function will create a semaphore and return the identifier. A new semaphore is created if key has the value IPC_PRIVATE or if no existing semaphore is associated to key and IPC_CREAT is asserted in semflg (i.e. semflg & IPC_CREAT isn't zero). The values of the semaphore will be set as sem_val.
- **int s_semop(struct s_sembuf *sop);**
  The function s_semop will perform an operation on a semaphore.
- **int s_semtimedop(struct s_sembuf *sop, struct timespec *timeout);**
  timed version of s_semop, which returns failure if it is unable to operation in time.
- **int s_semctl(int semid, int cmd, ...);**
  performs the control operation cmd on the semaphore semid.

Next.... data structures to be modified/implemented.

Data Structures

The Data structures that are part of the implementation are:

- **s_sem_array**
  The Kernel will be keeping the information of semaphores in this data structure:
- **s_sem_queue**
  This structure will be used to keep information about processes waiting on a semaphore
- **s_sembuf**
  This Structure will be used to represent semaphore operations to be done. The function, s_semop takes this structure as its argument.
- **s_sem_undo**
  This Structure keeps information about the undo operations that are to be done on a semaphore.
- **union s_semun**

  This Union is used as a parameter for the s_semctl function.

next..... functions to be implemented.

Functions to Implement

The internal functions of the implementation are

- `int is_newary (key_t key, int semflg);`
- `int sys_s_semget (key_t key, int semflg, int sem_val);`
- `int sys_s_semop (struct s_sembuf *sop);`
- `int sys_s_semtimedop (struct s_sembuf *sop, struct timespec *timeout);`
- `void s_append_to_queue (struct s_sem_array * sma, struct s_sem_queue * q);`
- `void s_prepend_to_queue (struct s_sem_array * sma, struct s_sem_queue * q);`
- `void s_remove_from_queue (struct s_sem_array * sma, struct s_sem_queue * q);`
- `struct s_sem_undo* s_freeundos(struct s_sem_array *sma, struct s_sem_undo* un);`
- `void s_update_queue (struct s_sem_array * sma);`
- `int s_sem_revalidate(int semid, struct s_sem_array* sma, short flg);`
- `int s_count_semncnt (struct s_sem_array * sma);`
- `int s_count_semzcnt (struct s_sem_array * sma);`
- `void s_freeary (int id);`
- `int s_semctl_nolock(int semid, int cmd, int version, union semun arg);`
- `int s_semctl_main(int semid, int cmd, int version, union semun arg);`
- `int s_semctl_down(int semid, int cmd, int version, union semun arg);`
- `int sys_s_semctl (int semid, int cmd, union semun arg);`
- `int s_alloc_undo(struct s_sem_array *sma, struct s_sem_undo** unp, int semid, int alter);`
- `void s_sem_exit (void);`

END

Verification Tests

Verification Tests

- Extent Of Verification:

Verification Tests

- Extent Of Verification:

  A. CONFORMANCE TESTS:
  To test both that the header files are correct and that the behavior defined in the specification is implemented.

Verification Tests

- Extent Of Verification:

  A. CONFORMANCE TESTS:
  To test both that the header files are correct and that the behavior defined in the specification is implemented.

  B. FUNCTIONAL TESTS:
  To test the presence and behavior of all necessary functional areas of the specification.

Verification Tests

- Extent Of Verification:

  A. CONFORMANCE TESTS:
  To test both that the header files are correct and that the behavior defined in the specification is implemented.

  B. FUNCTIONAL TESTS:
  To test the presence and behavior of all necessary functional areas of the specification.

  C. STRESS TESTS:
  To monitor how the system behaves when it is taxed by excessively using the functional areas.

The Test Suite

The Test Suite

- Required Test Suite:
  - For each functional interface of the semaphore specification, there will be a corresponding script/program to test it extensively for conformance, completeness and stress-response.
  - Flexibility to run on one set of tests, or testing only one functional ares, will be needed.

The Test Suite

- Required Test Suite:
  - For each functional interface of the semaphore specification, there will be a corresponding script/program to test it extensively for conformance, completeness and stress-response.
  - Flexibility to run on one set of tests, or testing only one functional ares, will be needed.
- Available - OPEN POSIX TEST suite:
  - The Open Posix Test Suite is a widely accepted conformance benchmark for such IPC specifications.
  - Package: ‘posixtestsuite’ version 1.4.2.

Micro-Benchmarks for Performance

Micro-Benchmarks for Performance

- Performance measured using 2 basic micro-benchmarks:

Micro-Benchmarks for Performance

- Performance measured using 2 basic micro-benchmarks:
  I. FLIP TIME: Single process lock-unlock cycle.

  II. SWITCH TIME: Two processes using one semaphore.

Micro-Benchmarks for Performance

- Performance measured using 2 basic micro-benchmarks:

  I. FLIP TIME: Single process lock-unlock cycle.
     1. A single semaphore is created.

  II. SWITCH TIME: Two processes using one semaphore.

Micro-Benchmarks for Performance

- Performance measured using 2 basic micro-benchmarks:

I. FLIP TIME: Single process lock-unlock cycle.
   1. A single semaphore is created.
   2. A lock/unlock cycle is executed for K times.

II. SWITCH TIME: Two processes using one semaphore.

Micro-Benchmarks for Performance

- Performance measured using 2 basic micro-benchmarks:

  I. FLIP TIME: Single process lock-unlock cycle.
     1. A single semaphore is created.
     2. A lock/unlock cycle is executed for K times.
     3. The total time T, for K cycles is reported.

  II. SWITCH TIME: Two processes using one semaphore.

Micro-Benchmarks for Performance

- Performance measured using 2 basic micro-benchmarks:

I. FLIP TIME: Single process lock-unlock cycle.

1. A single semaphore is created.
2. A lock/unlock cycle is executed for K times.
3. The total time T, for K cycles is reported.

- FLIP TIME $\boxed{Tf = T/2K}$

II. SWITCH TIME: Two processes using one semaphore.

Micro-Benchmarks for Performance

- Performance measured using 2 basic micro-benchmarks:

I. FLIP TIME: Single process lock-unlock cycle.
  1. A single semaphore is created.
  2. A lock/unlock cycle is executed for K times.
  3. The total time T, for K cycles is reported.
  - FLIP TIME $\boxed{Tf = T/2K}$

II. SWITCH TIME: Two processes using one semaphore.
  1. A process creates two semaphores,

Micro-Benchmarks for Performance

- Performance measured using 2 basic micro-benchmarks:

I. FLIP TIME: Single process lock-unlock cycle.
   1. A single semaphore is created.
   2. A lock/unlock cycle is executed for K times.
   3. The total time T, for K cycles is reported.
   - FLIP TIME $\boxed{Tf = T/2K}$

II. SWITCH TIME: Two processes using one semaphore.
   1. A process creates two semaphores,
   2. Process locks 1st, unlocks 2nd and waits on 1st.

Micro-Benchmarks for Performance

- Performance measured using 2 basic micro-benchmarks:

I. FLIP TIME: Single process lock-unlock cycle.

1. A single semaphore is created.
2. A lock/unlock cycle is executed for K times.
3. The total time T, for K cycles is reported.

- FLIP TIME $\boxed{Tf = T/2K}$

II. SWITCH TIME: Two processes using one semaphore.

1. A process creates two semaphores,
2. Process locks 1st, unlocks 2nd and waits on 1st.
3. A second process starts, waits on 1st and clears 2nd.

Micro-Benchmarks for Performance

- Performance measured using 2 basic micro-benchmarks:

I. FLIP TIME: Single process lock-unlock cycle.
   1. A single semaphore is created.
   2. A lock/unlock cycle is executed for K times.
   3. The total time T, for K cycles is reported.
   - FLIP TIME $\boxed{Tf = T/2K}$

II. SWITCH TIME: Two processes using one semaphore.
   1. A process creates two semaphores,
   2. Process locks 1st, unlocks 2nd and waits on 1st.
   3. A second process starts, waits on 1st and clears 2nd.
   4. The total time T, for K cycles is reported.

Micro-Benchmarks for Performance

- Performance measured using 2 basic micro-benchmarks:

I. FLIP TIME: Single process lock-unlock cycle.
   1. A single semaphore is created.
   2. A lock/unlock cycle is executed for K times.
   3. The total time T, for K cycles is reported.
   - FLIP TIME $\boxed{Tf = T/2K}$

II. SWITCH TIME: Two processes using one semaphore.
   1. A process creates two semaphores,
   2. Process locks 1st, unlocks 2nd and waits on 1st.
   3. A second process starts, waits on 1st and clears 2nd.
   4. The total time T, for K cycles is reported.
   5. Each cycle = 4 semaphore flips + 2 context switches.

Micro-Benchmarks for Performance

- Performance measured using 2 basic micro-benchmarks:

I. FLIP TIME: Single process lock-unlock cycle.
   1. A single semaphore is created.
   2. A lock/unlock cycle is executed for K times.
   3. The total time T, for K cycles is reported.
   - FLIP TIME $\boxed{Tf = T/2K}$

II. SWITCH TIME: Two processes using one semaphore.
   1. A process creates two semaphores,
   2. Process locks 1st, unlocks 2nd and waits on 1st.
   3. A second process starts, waits on 1st and clears 2nd.
   4. The total time T, for K cycles is reported.
   5. Each cycle = 4 semaphore flips + 2 context switches.
   - SWITCH TIME $\boxed{Ts = (T - 4.K.Tf)/(2.K)}$

Performance Tests

Performance Tests

- Both tests should be run in 3 load conditions:

Performance Tests

- Both tests should be run in 3 load conditions:
  1. Stand-alone. (say time *Ta*)

Performance Tests

- Both tests should be run in 3 load conditions:
  1. Stand-alone. (say time *Ta*)
  2. CPU load at lower priority. (say time *Tb*)

Performance Tests

- Both tests should be run in 3 load conditions:
  1. Stand-alone. (say time Ta)
  2. CPU load at lower priority. (say time Tb)
  3. CPU load at equal priority. (say time Tc)

Performance Tests

- Both tests should be run in 3 load conditions:
  1. Stand-alone. (say time Ta)
  2. CPU load at lower priority. (say time Tb)
  3. CPU load at equal priority. (say time Tc)
- Shows behaviour of OS scheduler. Ensure $\boxed{Ta < Tb < Tc}$

Performance Tests

- Both tests should be run in 3 load conditions:
  1. Stand-alone. (say time Ta)
  2. CPU load at lower priority. (say time Tb)
  3. CPU load at equal priority. (say time Tc)
- Shows behaviour of OS scheduler. Ensure $\boxed{Ta < Tb < Tc}$
- Run above tests for old and new implementations.

Performance Tests

- Both tests should be run in 3 load conditions:
  1. Stand-alone. (say time Ta)
  2. CPU load at lower priority. (say time Tb)
  3. CPU load at equal priority. (say time Tc)
- Shows behaviour of OS scheduler. Ensure $\boxed{Ta < Tb < Tc}$
- Run above tests for old and new implementations.
  - Ta-old: Tf, Ts
  - Tb-old: Tf, Ts
  - Tc-old: Tf, Ts

Performance Tests

- Both tests should be run in 3 load conditions:
  1. Stand-alone. (say time Ta)
  2. CPU load at lower priority. (say time Tb)
  3. CPU load at equal priority. (say time Tc)
- Shows behaviour of OS scheduler. Ensure $\boxed{Ta < Tb < Tc}$
- Run above tests for old and new implementations.
  - Ta-old: Tf, Ts
  - Tb-old: Tf, Ts
  - Tc-old: Tf, Ts
  - Ta-new: Tf, Ts
  - Tb-new: Tf, Ts
  - Tc-new: Tf, Ts

Performance Tests

- Both tests should be run in 3 load conditions:
  1. Stand-alone. (say time Ta)
  2. CPU load at lower priority. (say time Tb)
  3. CPU load at equal priority. (say time Tc)
- Shows behaviour of OS scheduler. Ensure $\boxed{Ta < Tb < Tc}$
- Run above tests for old and new implementations.
  - Ta-old: Tf, Ts
  - Tb-old: Tf, Ts
  - Tc-old: Tf, Ts
  - Ta-new: Tf, Ts
  - Tb-new: Tf, Ts
  - Tc-new: Tf, Ts
- Compare and analyze performance results.

Plan for Implementation

Here is the plan for implementation the project.

No.	Project Stage	Resources	
1	Coding and debugging	5 days	3 persons
2	Integration of functionality	5 days	2 persons
3	Writing Test Suite	5 days	1 persons
4	Integration with Kernel	5 days	3 persons
5	Functional Testing	2 days	3 persons
6	Performance Analysis	2 days	3 persons
7	Packaging and manuals	1 days	3 persons

The complete coding to packaging will take 60 programmer days.

Conclusions

Conclusions

- The existing implementation of semaphores in Linux is heavyweight in memory usage and performance due to usage of 'sets of semaphores'

Conclusions

- The existing implementation of semaphores in Linux is heavyweight in memory usage and performance due to usage of 'sets of semaphores'
- A lightweight implementation of semaphores will involve a mechanism that will allow single semaphores to be associated with resources. We have here presented a detailed design and interface definition for such a lightweight semaphore. We have also outlined a plan for implementation.

Conclusions

- The existing implementation of semaphores in Linux is heavyweight in memory usage and performance due to usage of 'sets of semaphores'
- A lightweight implementation of semaphores will involve a mechanism that will allow single semaphores to be associated with resources. We have here presented a detailed design and interface definition for such a lightweight semaphore. We have also outlined a plan for implementation.
- The improvement in performance expected is difficult to quantify. However it is expected to be significant enough to justify proceeding with the implementation.

References

T. Jones, A. Koniges, and R. K. Yates.
Performance of the IBM general parallel file system.
pages 673–683.

Frank Schmuck and Roger Haskin.
GPFS: Shared-disk file system for large computing clusters.
In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, January 2002.

K. Zuberi and K. Shin.
An efficient semaphore implementation scheme for small-memory embedded systems.
pages 25–37.