IMPLEMENTING SYNCHRONIZATION USING PIPES

Ştefan Udrişţoiu, Cătălin Cerbulescu and Eugen Ganea

Faculty of Automatics, Computers and Electronics,
University of Craiova

Abstract: This paper describes how mutexes, semaphores and condition can be implemented in user space using pipes. The implementation is based on pipe's properties to block reading processes when trying to read from empty pipes and that a byte is red only once. The main asset of the implementation is its extreme simplicity.

Keywords: operating system, process synchronization.

1. INTRODUCTION

The need for process synchronization is so obvious that we won't insist on its importance. In this paper, we describe an implementation in user space of "classical" synchronization mechanisms: mutexes, semaphores and condition variables. In this implementation, the only operating system mechanism used is pipe.

Usually, synchronization is implemented in user space with busy waiting or using special hardware instructions such as TSL (Test and Set Lock). Disabling interrupts is out of the question, even in the case when this is possible outside the kernel, because an incorrect program will freeze the system. Busy waiting is not recommended because it wastes CPU time and it can lead to priority inversion. One of the best implementations using TSL is similar with that proposed the Tanenbaum(2001) for mutexes, synchronizing threads in user space:

```
mutex_lock:
   TSL REGISTER, MUTEX
   CMP REGISTER, #0
   JZE ok
   CALL sched_yield
   JMP mutex_lock
ok:
   RETI
```

In this solution we have replaced thread_yield, a call to the thread scheduler implemented in user space, with a system call to scheduler, sched_yield, which makes the calling process to give up the CPU.

The above solution is very efficient when the mutex is not taken because it doesn't need a system call. But in the opposite case, the processor is released and the process is put at the end of its corresponding ready list. If the process that owns the resource has a lower priority then we have priority inversion problem and only the scheduler can solve that. Even without priority inversion this implementation is inefficient. Consider the case when we have n processes waiting to lock the same mutex and the critical region of the process that has the mutex is q quantum "long", i.e. needs q quantum to execute. At the end of its quantum, the process that owns the mutex will give up the processor to the first waiting process, which is trying to lock the same mutex. The new running process, after executing a very short code, will decide to release CPU voluntarily to the next waiting process, which is not different at all from the previous one and so is the outcome. After the owning process releases the CPU we actually have a rapid succession of n context switches, until the running process will be again the one that is the only real ready process and this sequence of context switches will repeat unnecessarily until the owning process will exit the critical region.

2. IMPLEMENTATION

In our implementation will make use of pipe properties:

1. If the pipe is empty, the process trying to read from will sleep until another process writes data into the pipe, at which time all sleeping processes that were waiting for data wake up and race to read the pipe, (Bach ,1986).
2. Every byte from pipe will be read only once.

2.1 How do we recognize a good solution?

A solution to the critical-section problem must satisfy the following three requirements (Silberschatz et al., 2002):

1. Mutual Exclusion: When a process is executing in its critical section, then no other processes can be executing in their critical sections.
2. Progress: If no process is executing in its critical section and some processes wish to enter in their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. Bounded Waiting: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Another requirement is not to make any assumption about the speed of CPUs.

2.2 Non-recursive and non-error checking mutex implementation.

We define mutexes as a structure containing both ends of a pipe:

```
typedef struct {
      int pipes[2];
} Mutex;
```

To create a mutex we use the following call:

```
int createMutex(Mutex* mutex ) {
      int rez = pipe(mutex->pipes);
      if (rez == -1) {
            return -1;
      }
      write(mutex->pipes[1], "1",
      1);
      return 1;
}
```

Mutex consists in creating a pipe and writing a single byte in that.

The beginning of critical section is marked with the take_mutex call:

```
int take_mutex(Mutex mutex) {
      char b;
      int rez =
      read(mutex.pipes[0], &b, 1);
      return rez;
}
```

The algorithm is very simple: it tries to read a byte from the pipe. If the pipe is empty, the process issuing the call blocks until some other process writes into the pipe. Remember that a newly created mutex has a byte wrote in its pipe thus the mutex being not taken. When a process takes a free mutex it drains the pipe, blocking the following processes trying to obtain the mutex.

When a process releases a mutex writes a single octet into mutex's pipe:

```
int release_mutex(Mutex mutex) {
      int rez =
      write(mutex.pipes[1], "1",
      1);
      return rez;
}
```

If two processes are waiting to take the mutex they will be waked-up but only one will succeed to take the mutex because there is only one byte in the pipe.

2.3 Error check mutex implementation.

In this type of mutex, if the same process tries to lock a mutex which already owns an error is returned. In the later implementation the process deadlock itself. The modifications are minimal:

```
#define EDEADLK -2
typedef struct {
      int pipes[2];
      int owner;
} Mutex;
```

Initially, nobody will own the mutex:

```
int createMutex(Mutex* mutex ) {
      int rez = pipe(mutex->pipes);
      if (rez == -1) {
            return -1;
      }
      write(mutex->pipes[1], "1",
      1);
      mutex->owner = -1;
      return 1;
}
```

When trying to lock the mutex the process checks if it already owns the mutex:

```
int take_mutex(Mutex* mutex) {
      char b;
      if (mutex->owner ==
            getpid()){

            return EDEADLK;
      }
      int rez = read(mutex
            ->pipes[0], &b, 1);
      if (rez != -1) {
            mutex->owner = getpid();
      }
      return rez;
}
```

After successfully locks the mutex the process marks itself as the owner of the mutex. If a process inherits the mutex after it was locked by the parent, the mutex owner will be marked by the parent id. If the mutex is inherited before was locked the owner will be –1. The field is used for the process which wants to lock the mutex as an indicator for the case in which the calling process is already the owner. It doesn't matter if the mutex is free or owned by another process.

When it releases a mutex the process first checks to see if it is the owner. If true, writes the magic byte and marks for itself that the mutex its free:

```
int release_mutex(Mutex* mutex) {
      if (mutex->owner !=
            getpid()){

            return -3;
      }
      int rez = write(mutex
            ->pipes[1], "1", 1);
      if (rez != -1) {
            mutex->owner = -1;
      }
      return rez;
}
```

2.4 Recursive mutex implementation.

The difference from the previous case is that the process is allowed to retake a mutex. A mutex locked n times has to be released the same number of times. So, besides the owner we have to count the locks:

```
typedef struct {
      int pipes[2];
      int owner;
      int locks;
} Mutex;
```

The creation of a mutex is the same as in the error-checking mutex. The take and release are slightly different:

```
int take_mutex(Mutex* mutex) {
      char b;
      if (mutex->owner ==
            getpid()){

            mutex->locks++;
            return mutex->locks;
      }
      int rez = read(mutex
            ->pipes[0], &b, 1);
      if (rez != -1) {
            mutex->owner = getpid();
            mutex->locks = 1;
      }
      return rez;
}
```

If the calling process already owns the mutex then all it does is locks increment. If not, it enters the race for mutex by reading from the pipe. After a successful read it marks the mutex as owned and initializes the number of locks.

At mutex release, after checking if the calling process is the owner of the mutex, the number of locks is decremented. The mutex is actually released only when the number of locks reaches zero:

```
int release_mutex(Mutex* mutex) {
      if (mutex->owner !=
            getpid()){

            return -3;
      }
      mutex->locks--;
      if (mutex->locks > 0) {
            return mutex->locks;
      }

      int rez = write(mutex
            ->pipes[1], "1", 1);
      if (rez != -1) {
            mutex->owner = -1;
      }
      return rez;
}
```

2.5 Semaphore implementation.

Semaphores are a small variation of the first mutexes' implementation:

```
typedef struct {
      int pipes[2];
} Semaphore;
```

When a semaphore is created its state is initialized with a given value. The initialization is made by writing state bytes in semaphore's pipe:

```
int createSemaphore(Semaphore*
      semaphore, int state) {
      int rez = pipe(semaphore
            ->pipes);
      if (rez == -1) {
            return -1;
      }
      while(state > 0) {
            write(semaphore
            ->pipes[1], "1", 1);
            state--;
      }
      return 1;
}
```

Operations on semaphores are very simple: post consists in writing a byte and wait reading a byte.

```
int waitSemaphore(Semaphore*
   semaphore) {
   char b;
   int rez = read(semaphore->
            pipes[0], &b, 1);
   return rez;
}

int postSemaphore(Semaphore*
   semaphore) {
   int rez = write(semaphore->
            pipes[1], "1", 1);
   return rez;
}
```

2.5 Condition variables implementation.

The implementation of condition variables is slightly more complicated. Will start with the structure wich consists of two pipes, one for blocking and one for counting waiting processes:

```
typedef struct {
      int waitPipe[2];
      int countPipe[2];
} ConditionVariable;
```

When creating a condition variable in its count pipe is wrote a '0' to keep that pipe non empty:

```
int createCondition(
      ConditionVariable* cond) {

      int rez = pipe(cond
            ->countPipe);
      if (rez == -1) {
            return -1;
      }
      rez = pipe(cond
            ->waitPipe);
      if (rez == -1) {
            return -1;
      }
      rez = write(cond->
            countPipe[1], "0", 1);
      if (rez == -1) {
            return -1;
      }
      return 1;
}
```

We wrote a '0' into the count pipe because we assume the most unfavorable case that we don't have possibility to do an unblocking read from pipe.

The implementation of wait is:

```
int waitForCondition(
      ConditionVariable* cond,
      Mutex* mutex) {

      char b;
      release_mutex(mutex);
      write(cond->countPipe[1],
            "1", 1);
      read(cond->waitPipe[0],
            &b, 1);
      take_mutex(mutex);
      return 1;
}
```

After we release the guarding mutex we increase the count of waiting processes, writing a '1' in the count pipe, and we block reading from wait pipe. After the read, which should unblock after receiving a signal we retake the mutex before ending the wait call.

When signaling a condition variable we must first check if there is at least one process waiting, i.e. at least a '1' in the count pipe:

```
#define BUFF_SIZE 1024
int signalCondition(
      ConditionVariable cond) {

      char buff[BUFF_SIZE];
      int i;
      int n = read(cond
            ->countPipe[0], buff,
            BUFF_SIZE);
      for (i = 0; i < n; i++) {
            if (buff[i] == '1') {
                  buff[i] = '0';
                  write(cond->
                  waitPipe[1],
                  "1", 1);
                  write(cond->
                  countPipe[1],
                  &buff[i], n - i);
                  return 1;
            }
      }
```

```
      write(cond->countPipe[1],
      "0", 1);
      return 0;
}
```

After signaling one waiting process we restore the content of count pipe, after a "decrement". In the case when we have no waiting process we still write in the count pipe a '0' just to keep it nonempty.

The broadcast is simpler, we write a byte in the wait pipe for every '1' found in the count pipe, i.e. for every waiting process:

```
int broadcastCondition(
      ConditionVariable cond) {

      char buff[BUFF_SIZE];
      int i;
      int n = read(cond
            ->countPipe[0], buff,
            BUFF_SIZE);
      for (i = 0; i < n; i++) {
            if (buff[i] == '1') {
                  write(cond->
                  waitPipe[1],
                  "1", 1);
            }
      }
      write(cond
      ->countPipe[1], "0", 1);
      return 0;
}
```

After signaling all waiting processes we write a '0' in the count pipe. The count pipe must be kept non-empty because the operation of signaling a variable is non-blocking.

3. CONCLUSION

The solutions presented are very simple and they necessitate only a single, very common, operating system mechanism, the pipes. The disadvantage over a kernel implementation can come from pipe implementation: usually, when a process writes into a pipe then all processes blocked when reading are waked up. Using named pipes the synchronization can be used also by unrelated processes. The events can be implemented in a similar mode with condition variables.

REFERENCES

Bach, M.J. (1986). *The Design of the UNIX Operating System*. Prentice Hall.

Silberschatz, A., P. Baer Galvin and G. Gagne (2002). *Operating System Concepts*. John Willey & Sons.

Tanenbaum, A.S. (2001). *Modern Operating System*. Prentice Hall.