Process Synchronization (III)

September 20, 2017

Reader-Writer Problem

- A data set is shared among a number of concurrent processes
  - Readers: only read the data set; they do **not** perform any updates
  - Writers: can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at a time.

Reader-Writer Problem: Analogies

It is fine for a group of people to watch (read) a TV simultaneously.

But changing (write) programs must be done sequentially (no concurrency)!

Reader-Writer Problem: Observations

- A writer is allowed to access the data set only when there is no other writer or reader accessing

- Readers can simultaneously access the data set
  - When a reader arrives, it is allowed to access the data set if (i) no one is accessing the data set, or (ii) some other reader is reading; otherwise (some writer is writing) , it should wait.
  - For a waiting reader, it is allowed to read if the writing writer completes.

Reader-Writer Problem: Observations → Solution

- A writer is allowed to access the data set only when there is no other writer or reader accessing

- Data Structure
  - Semaphore wrt initialized to 1 → to ensure at most one writer (no 2+ writers or one writer and reader(s)) allowed to access the data set (To guard access to data set)
- Code: The structure of a writer process

```
wait (wrt) ;
….  writing is performed …
signal (wrt) ;
```

Reader-Writer Problem: Observations → Solution

- Readers can simultaneously access the data set
  - When a reader arrives, it is allowed to access the data set if (i) no one is accessing the data set, or (ii) some other reader is reading; otherwise (some writer is writing) , it should wait.
  - For a waiting reader, it is allowed to read if the writing writer completes.

Additional Data Structures:

- Integer readcount initialized to 0 → to keep track of number of readers that are currently reading the data set
- Semaphore mutex initialized to 1 → to ensure mutual exclusive modification of readcount (To guard access to readcount)

Readers-Writers Problem: Solution

- The structure of a reader process

```
wait (mutex) ;
if (readcount == 0)  wait (wrt) ;
    //check if there is already reader being reading
    //if yes, go ahead to read;
    //otherwise, go on only if no writer writing
    //note: if this reader has to wait, follow-up readers should wait too; so block them
readcount ++ ;
signal (mutex);
                ... reading is performed...
wait (mutex) ;
readcount  - - ;
if (readcount  == 0)  signal (wrt) ; //if this is the last reader
signal (mutex) ;
```

Semaphore Implementation

- Must guarantee:
  - No two processes can actively execute wait( ) on the same semaphore at the same time (but when one is blocked by wait, another is allowed to execute it)
  - No two processes can modify the value of semaphore at the same time (i.e., signal( ) and "S- -" of wait( ) must be atomic)
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.

Semaphore Implementation

All competing processes share a variable lock initialized to false

```
Signal(S) {
        while(testAndSet(&lock));
        S++;
        lock=False;
}
```

Semaphore Implementation

All competing processes share a variable lock initialized to false

```
wait(S) {
        while(testAndSet(&lock));
        while (S < =0) {
                lock=False;
                while(testAndSet(&lock);
        }
        S - -;
        lock=False;
}
```

Semaphore Implementation

- Ideally, will not have busy waiting in critical section
  - Otherwise, applications may spend lots of time in critical sections.

Semaphore Implementation with no Busy waiting

- With each semaphore, there are
  - value (of type integer)
  - list: an associated waiting queue. Each entry in a waiting queue has two items: (1) pointer to a process/thread; (2) pointer to next record in the list
- Two (atomic) operations:
  - block – place the process invoking the operation on a waiting queue.
  - wakeup – remove one of processes on a waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting

- (Rewritten) Definition of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

Semaphore Implementation with no Busy waiting

- Implementation of wait (example):

```
wait(semaphore *S) {
        while(testAndSet(&lock));
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                lock=False;
                block();
        }
        else lock=False;
}
```

Semaphore Implementation with no Busy waiting

- Definition of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Semaphore Implementation with no Busy waiting

- Implementation of signal (example):

```
signal(semaphore *S) {
        while(testAndSet(&lock));
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
        lock=False;
}
```