Process Synchronization (II)

Semaphore

- An abstract data type
  - Semaphore *S* – integer variable
  - Two standard operations to access S: wait( ) and signal( )
    - Originally called P( ) and V( )
- Can only be accessed via two (atomic) operations

```
wait (S) {
        while (S <= 0); //blocked
        S--;
}
signal (S) {
      S++;
}
```

Usage of Semaphores

- Binary semaphore – integer value S can range only between 0 and 1
  - Also known as mutex (i.e., mutual exclusive) locks
  - Provides mutual exclusion (the following is pseudo code)

  Semaphore mutex;   // initialized to 1

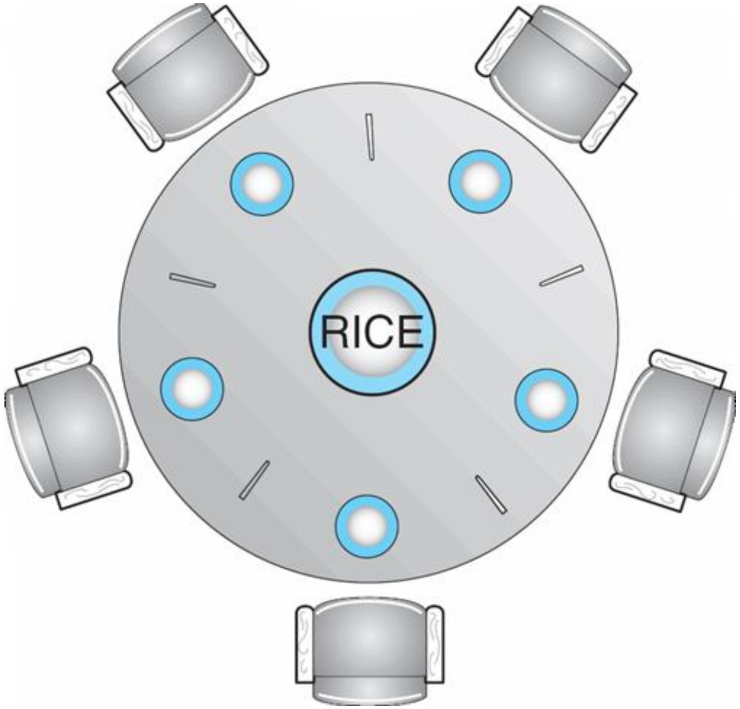
  wait (mutex);

  … Critical Section …

  signal (mutex);

  … remainder section …
- Counting semaphore – integer value can range over an unrestricted domain

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem

- The structure of Philosopher *i*:

```
wait ( chopstick[i] );
wait ( chopStick[ (i + 1) % 5] );
        … eat…
signal ( chopstick[i] );
signal (chopstick[ (i + 1) % 5] );
        …  think…
```

POSIX Semaphores

- Two types of semaphores
  - Unnamed semaphores
    - Volatile
    - Storage is allocated by user program
  - Named semaphores
    - Permanent (like named Pipe (FIFO))
    - Storage is allocated by the OS
  - Defined in <semaphore.h>
  - Provided by Linux (You can use it on pyrite)

Unnamed Semaphores

- Initialize a semaphore:

  int sem_init(sem_t *sem, int pshared, unsigned value)

  - sem: pointer to a sem_t structure allocated by user program
  - pshared=0 if the sem is shared by threads in the same proc; otherwise, the sem can be shared by multiple processes (the sem_t structure should be allocated in the memory block shared by these processes)
  - value: initial value of the sem
- Destroy a semaphore:

  int sem_destory(sem_t *sem)

Unnamed Semaphores

- Wait on a semaphore:

    int sem_wait(sem_t *sem)

- Signal a semaphore:

    int sem_post(sem_t *sem)

Named Semaphores

- Create and open a semaphore:

  sem_t *sem_open(char *name, O_CREAT, mode_t mode, unsigned value)

  - name: a permanent name (like a file name) for the sem
  - mode: permission mode: combination of S_IRUSR (owner read), S_IWUSR (owner write), S_IXUSR (owner exec), S_IRGRP (owner's group read), S_IWGRP (group write), S_IXGRP (group exec), S_IROTH (other users read), S_IWOTH (other write), S_IXOTH (other exec)
  - value: initial value of the sem
  - Return the pointer to the sem_t structure of the sem
- Open an existing semaphore:

  sem_t *sem_open(char *name, O_EXCL)

Named Semaphores

- Once a named semaphore has been open, it can be accessed using sem_wait( ) and sem_post( ) as an unnamed semaphore
- Close a semaphore:

  sem_close(sem_t *sem);
- Destroy (permanently delete) a semaphore:

  sem_unlink(char *name);

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Two types of processes: producer processes and consumer processes
- Requirements:
  - No two processes can access the buffers simultaneously (mutual exclusion)
  - A producer cannot put an item to the buffers if the buffers are all full
  - A consumer cannot remove an item from the buffer if the buffers are all empty

Shared Variables:

int numAvailBuffer=N;

semaphore mutex; //=1, to ensure mutual exclusion

Producer:

```
wait(mutex);
while(1){
        if(numAvailBuffer>0){
                numAvailBuffer --;
                … put data into a buffer …
                signal(mutex);
                break;
        }else{
                signal(mutex);
                wait(mutex);
        }
}
```

Consumer:

```
wait(mutex);
while(1){
        if(numAvailBuffer<N){
                numAvailBuffer ++;
                … fetch data from a buffer …
                signal(mutex);
                break;
        }else{
                signal(mutex);
                wait(mutex);
        }
}
```

Bounded-Buffer Problem

- To ensure mutual exclusion
  - Semaphore mutex initialized to the value 1 (number of admissible process)
  - Only when mutex == 1 can a process proceed to access the buffers (i.e., put in or remove an item)
- To synchronize consumer processes
  - Semaphore full initialized to the value 0 (number of full buffers)
  - Only when full >0 can a consumer process proceed
- To synchronize producer processes
  - Semaphore empty initialized to the value N (number of empty buffers)
  - Only when empty >0 can a producer process proceed

Bounded Buffer Problem

- The structure of the producer process

```
… produce an item
wait (empty);
wait (mutex);
… add the item to the  buffer
signal (mutex);
signal (full);
```

Bounded Buffer Problem

- The structure of the consumer process

```
wait (full);
wait (mutex);
… remove an item from  buffer
signal (mutex);
signal (empty);
 … consume the item in nextc
```

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 the same 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 as well; so block them
readcount ++ ;
signal (mutex);
                    … reading is performed…
```

Readers-Writers Problem: Solution

- The structure of a reader process

… 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;
}
```