System Programming

10. POSIX Semaphores

Seung-Ho Lim
Dept. of Computer & Electronic Systems Eng.

Semaphore

- **System calls designed by Dijkstra in 1960's**
- **Used to control the critical section (i.e. resource sharing) b/w multiple threads(processes)**
- **Basic features**
  - A ***block/wakeup algorithm*** for mutual exclusion.
  - If a process cannot enter a critical section, the process will be ***blocked.*** (release CPU)
  - When the owner process exits a critical section, it ***wakes the waiting process up.*** (make it to be ***ready***)
  - No waste of CPU time.
  - A ***waiting queue*** is necessary if several processes are waiting for the permissions for entering their critical sections.

Counting Semaphore

```
class Semaphore
{   private:
    int value;
    PCB *queue;
}

Semaphore::Semaphore (n)
{  value = n; }

Semaphore::wait()
// original name: "P" operation
{
    value--;
    if ( value < 0 ) {
        block the calling process;
        add it to the wait queue of this
           semaphore;
    }
}
```

```
Semaphore::signal()
 // original name: "V" operation
{
      value++;
      if ( value <= 0 ) {
          remove the first process from the
             wait queue;
          add it to the scheduling queue;
      }
}
```

Binary Semaphore

```
class Semaphore
{ private:
    int value;
    PCB *queue;
}

Semaphore::Semaphore (1)
{  value = 1; }

Semaphore::wait()
     // original name: "P " operation
{
    if (value == 1) value = 0;
    else {
          block the calling process;
          add it to the wait queue of this
                semaphore;
    }
}
```

```
Semaphore::signal()
    // original name: "V " operation
{
    if ( queue is not empty) {
         remove the first process from the wait
              queue;
         add it to the scheduling queue;
    } else
         value = 1;
}
```

Binary semaphore is a special case of Counting semaphore!

Mutual Exclusion by Semaphore

- Mutual exclusion using **a binary or a counting semaphore**

Semaphore S(**1**); // global variable for mutual exclusion, initialized with **1**

```
process/thread  A
-----------------------------
 S.wait ();
-----------------------------
     critical section;
-----------------------------
 S.signal ();
-----------------------------
remainder section;
```

```
process/thread  B
-----------------------------
S.wait ();
-----------------------------
   critical section;
-----------------------------
S.signal ();
-----------------------------
remainder section;
```

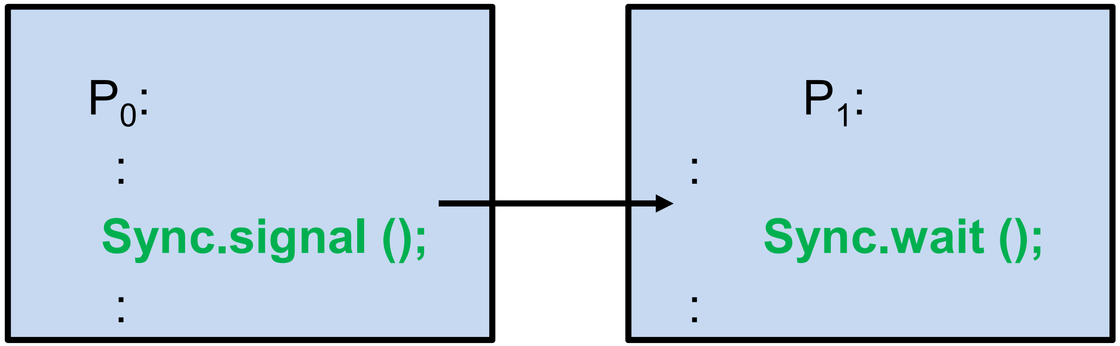
Resource Allocation by a Counting Semaphore

Semaphore Printer(**3**); // in case that the system has 3 identical printers.

```
      :
----------------------
 Printer.wait();
----------------------
    use a printer;  // for a mutually exclusive use of a printer
----------------------
 Printer.signal();
----------------------
    remainder section;
```

Synchronization with a Semaphore

Semaphore Sync(0); // global var. with the initial value of 0



- In the case that the P_1 reaches the wait-point earlier than P_0's signal point
  - P_1 will be blocked until P_0 reaches the signal point because the semaphore value is 0.
- In the case that the P_0 reaches the signal-point earlier than P_1's wait-point
  - P_0 will continue without waiting because the semaphore value is already 1.
- This means that P_0 can proceed after P_1's sync-point.

Producer/Consumer with a Counting Semaphore

```
// global variables
Semaphore mutex(1), num_buffer(n), num_data(0);
int rear = front = -1; count = 0;
```

```
while (1)  // producer
{
    produce an item;
    num_buffer.wait ();  // for sync
    mutex.wait();
    buffer[rear] = pdata;
    rear = (rear + 1) % n ;
    mutex.signal();
    num_data.signal(); // for sync
}
```

```
while(1) // consumer
{
    num_data.wait(); // for sync
    mutex.wait();
    cdata = buffer[front] ;
    front = (front + 1) % n ;
    mutex.signal();
    num_buffer.signal (); // for sync
    process an item;
}
```

1. "Count" variable is unnecessary because we use a counting semaphores ! Counting semaphore ***num_data*** itself has the number of data.
2. For a single producer and a single consumer, ***mutex.wait()*** and ***mutex.signal()*** is unnecessary.

POSIX SEMAPHORES

- NAME

  sem_init, sem_wait, sem_trywait, sem_post, sem_getvalue, sem_destroy - operations on semaphores

- SYNOPSIS

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t * sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t * sem);
int sem_getvalue(sem_t * sem, int * sval);
int sem_destroy(sem_t * sem);
```

- *must be linked with pthread library.*

Semaphore intialization

```
#include <semaphore.h>

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

- parameters:
  - *sem* : initializes the semaphore object pointed to by sem.
  - *pshared* :
    - 0 means the semaphore is local to the current process (when it iss zero)
    - non-zero means that it is shared between several processes (should be located in a region of shared memory b/w the processes)
  - *value* : initial value for the semaphore

Semaphore wait operations

```
#include <semaphore.h>

int sem_wait(sem_t * sem);
int sem_trywait(sem_t *sem);
```

- sem_wait()
  - suspends the calling thread until the semaphore pointed to by sem has non-zero count.
  - It then atomically decreases the semaphore count.
- sem_trywait()
  - is a non-blocking version of sem_wait.
  - If the semaphore pointed to by sem has non-zero count, the count is atomically decreased and sem_trywait immediately returns 0.
  - If the semaphore count is zero, sem_trywait immediately returns with error EAGAIN.

Semaphore post(signal) operations

```
#include <semaphore.h>

int sem_post(sem_t * sem);
int sem_getvalue(sem_t * sem, int * sval);
```

- In POSIX, semaphore signal operation is named ***post***
- sem_post()
  - atomically increases the count of the semaphore pointed to by *sem*.
  - this function never blocks and can safely be used in asynchronous signal handlers.
- sem_getvalue()
  - stores the current count of the semaphore *sem* in the location pointed to by sval

Semaphore destruction

```
#include <semaphore.h>

int sem_destroy(sem_t * sem);
```

- destroys a semaphore object pointed by *sem*
  - frees the resources it might hold.
  - no threads should be waiting on the semaphore at the time `sem_destroy` is called.
  - destroying a semaphore that other processes or threads are currently blocked on produces undefined behavior

Example : Mutual Exclusion using Semaphore (1)

semaphore.c

```
#include <semaphore.h>
int cnt=0;
static sem_t hsem;

int main(int argc, char *argv[])
{       pthread_t thread1;
        pthread_t thread2;

        if (sem_init(&hsem, 0, 1) < 0){
                fprintf(stderr, "Semaphore Initilization Error\n");
                return 1;
        }
        pthread_create(&thread1, NULL, Thread1, NULL);
        pthread_create(&thread2, NULL, Thread2, NULL);
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        printf("%d\n", cnt);
        sem_destroy(&hsem);

        return 0;
}
```

Example : Mutual Exclusion using Semaphore (2)

semaphore.c

```
void *Thread1(void *arg)
{
        int i, tmp;

        for(i=0; i<1000; i++){
            sem_wait(&hsem);
            tmp=cnt;
            usleep(1000);
            cnt=tmp+1;
            sem_post(&hsem);
        }
        printf("Thread1 End\n");

        return NULL;
}
```

```
void *Thread2(void *arg)
{
        int i, tmp;

        for(i=0; i<1000; i++){
            sem_wait(&hsem);
            tmp=cnt;
            usleep(1000);
            cnt=tmp+1;
            sem_post(&hsem);
        }
        printf("Thread2 End\n");

        return NULL;
}
```

Example: Producer & Consumer (1)

prod-cons.c

```
.......
#define MAX_BSIZE   10
int cnt=0;
static sem_t hsem, num_buff, num_data;

void *Producer(void *arg) // producer
{   int i, tmp;

    for(;;) {
        sem_wait(&num_buff);
        sem_wait(&hsem);
        cnt++;
        printf("prod cnt: %d \n", cnt);
        sleep(1);
        sem_post(&hsem);
        sem_post(&num_data);
    }
    printf("Producer Ends\n");
    return NULL;
}
```

Example: Producer & Consumer (2)

```
void *Consumer(void *arg) // consumer
{
    int i, tmp;

    for(;;) {
        sem_wait(&num_data);
        sem_wait(&hsem);
        cnt--;
        printf("cons cnt: %d \n", cnt);
        sleep(1);
        sem_post(&hsem);
        sem_post(&num_buff);
    }
     printf("Consumer Ends\n");
    return NULL;
}
```

Example: Producer & Consumer (3)

```
int main(int argc, char *argv[])
{   pthread_t thread1;
    pthread_t thread2;

    if (sem_init(&hsem, 0, 1) < 0){
        fprintf(stderr, "Semaphore Initialization Error\n");
        return 1;
    }
    if (sem_init(&num_buff, 0, MAX_BSIZE) < 0){
        fprintf(stderr, "Semaphore Initialization Error\n");
        return 1;
    }
    if (sem_init(&num_data, 0, 0) < 0){
        fprintf(stderr, "Semaphore Initialization Error\n");
        return 1;
    }
    pthread_create(&thread1, NULL, Producer, NULL);
    pthread_create(&thread2, NULL, Consumer, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("%d\n", cnt);
    sem_destroy(&hsem);

    return 0;
}
```