INF-2201
06 – Semaphores

John Markus Bjørndalen
2024

Synchronization recap

- Relatively easy to get a decent solution if no preemption
  - Can turn off interrupts to turn a preemptive to a nonpreemtive environment
  - Move synchronization to the kernel if necessary
- If preemption:
  - A correct solution that doesn't use hardware support quickly gets a) complicated and b) hard to prove and c) use spinning
  - Solutions that use hardware features (see figures on the right) are easier to reason about, but can still waste cycles while spinning
  - Priority inversion can be an issue

```
enter_region:
      TSL REGISTER,LOCK          | copy lock to register and set lock to 1
      CMP REGISTER,#0            | was lock zero?
      JNE enter_region           | if it was not zero, lock was set, so loop
      RET                        | return to caller; critical region entered

leave_region:
      MOVE LOCK,#0               | store a 0 in lock
      RET                        | return to caller
```

Figure 2-25. Entering and leaving a critical section using the TSL instruction.
Modern Operating Systems, Tanenbaum & Bos

```
enter_region:
      MOVE REGISTER,#1           | put a 1 in the register
      XCHG REGISTER,LOCK         | swap the contents of the register and lock variable
      CMP REGISTER,#0            | was lock zero?
      JNE enter_region           | if it was non zero, lock was set, so loop
      RET                        | return to caller; critical region entered

leave_region:
      MOVE LOCK,#0               | store a 0 in lock
      RET                        | return to caller
```

Figure 2-26. Entering and leaving a critical section using the XCHG instruction.
Modern Operating Systems, Tanenbaum & Bos

Easy solution 1 (for a single core OS)

- Provide a enter_region system call that
  - Enters the kernel
  - Disables interrupts
  - Runs the "try part" of enter_region on the right (don't use the loop)
  - If it doesn't succeed, blocks the thread/process and puts it on a waiting queue. Then run scheduler
  - If it succeeds, enable interrupts and return
- Provide a leave_region system call that
  - Enters the kernel
  - Disables the interrupts
  - Pops out one or more processes/threads from the waiting queue
  - Runs the leave_region bit on the right
  - Enables interrupts and returns
- High overhead

```
enter_region:
      TSL REGISTER,LOCK          | copy lock to register and set lock to 1
      CMP REGISTER,#0            | was lock zero?
      JNE enter_region           | if it was not zero, lock was set, so loop
      RET                        | return to caller; critical region entered

leave_region:
      MOVE LOCK,#0               | store a 0 in lock
      RET                        | return to caller
```

Figure 2-25. Entering and leaving a critical section using the TSL instruction.
Modern Operating Systems, Tanenbaum & Bos

```
enter_region:
      MOVE REGISTER,#1           | put a 1 in the register
      XCHG REGISTER,LOCK         | swap the contents of the register and lock variable
      CMP REGISTER,#0            | was lock zero?
      JNE enter_region           | if it was non zero, lock was set, so loop
      RET                        | return to caller; critical region entered

leave_region:
      MOVE LOCK,#0               | store a 0 in lock
      RET                        | return to caller
```

Figure 2-26. Entering and leaving a critical section using the XCHG instruction.
Modern Operating Systems, Tanenbaum & Bos

Producer-consumer problem

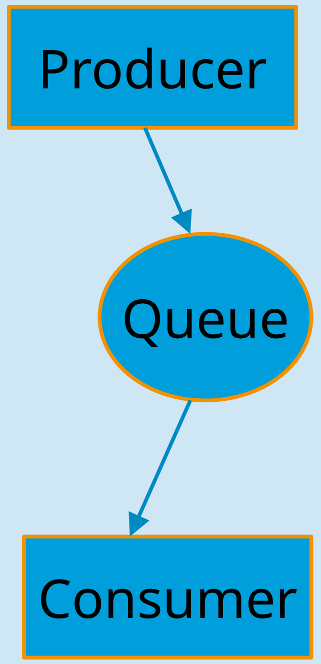
The producer-consumer problem has the followining parts

1) A producer: creates information that it adds to a queue. If the queue is full, it waits until there is room.
2) A queue (can be a circular buffer) with a limited number of slots (bounded buffer)
3) A consumer: waits until there is something in the queue, pulls the first item out and consumes it (ignore how).

The solutions has to observe the following:

- The producer and consumer are executing independently. You cannot assume anything about their speeds.
- The queue data structure must be preserved. A simple solution is to use a critical region around anything that handles the queue.

More general case: there might be multiple producers and consumers, but we will ignore that for now.



Producer-consumer – solution 1

- Solution that assumes only one producer and one consumer
- Both use sleep if they cannot continue adding or removing items from the queue
- The other end uses wakeup if they add or remove items
- Race condition… where?

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                item = produce_item( );         /* generate next item */
                if (count == N) sleep( );       /* if buffer is full, go to sleep */
                insert_item(item);              /* put item in buffer */
                count = count + 1;              /* increment count of items in buffer */
                if (count == 1) wakeup(consumer);       /* was buffer empty? */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                if (count == 0) sleep( );       /* if buffer is empty, got to sleep */
                item = remove_item( );          /* take item out of buffer */
                count = count − 1;              /* decrement count of items in buffer */
                if (count == N − 1) wakeup(producer);   /* was buffer full? */
                consume_item(item);             /* print item */
        }
}
```

Figure 2-27. Producer-consumer problem with a fatal race condition.
Modern Operating Systems, Tanenbaum & Bos

Producer-consumer – solution 1

- Race condition… where?
- A general tool for spotting potential race conditions
  - These are equivalent
    - Read-modify-write
    - Observe-decide-act
  - If the three are not done atomically (anbody can modify state between the steps), then there is a chance of a race condition
- Using the tool 1 (producer vs. consumer):
  - "if (count == N) sleep()" is an **observe** (read count), **decide** (compare count to N), and **act** (sleep).
  - There is no protection of the state, so the consumer may remove an item between reading count and the decide step in the producer.
    - The producer is not woken up even if there is room. May wake up the next time the consumer removes an item.
- Using the tool 2 (consumer vs. producer):
  - Observe from the point of the consumer
  - If this happens when the producer added the final item, the consumer may never wake up and consume it

```
#define N 100                                  /* number of slots in the buffer */
int count = 0;                                 /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                            /* repeat forever */
          item = produce_item( );              /* generate next item */
          if (count == N) sleep( );            /* if buffer is full, go to sleep */
          insert_item(item);                   /* put item in buffer */
          count = count + 1;                   /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);    /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                            /* repeat forever */
          if (count == 0) sleep( );            /* if buffer is empty, got to sleep */
          item = remove_item( );               /* take item out of buffer */
          count = count − 1;                   /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                  /* print item */
     }
}
```

Figure 2-27. Producer-consumer problem with a fatal race condition.
Modern Operating Systems, Tanenbaum & Bos

Semaphores

- Locks typically have two states:
  - 0: lock free / released
  - 1: lock taken / acquired
- A more general concept is a semaphore
  - General idea: use an integer to store the number of wakeups. Can be larger than 1!
  - Two operations (similar to acquire and release). Both are atomic:
    - **Down**: check if value is larger than 0. If it was 0, sleep / wait. When it is 1 or larger: count down by one.
    - **Up**: add one to the semaphore. If there is a waiting process, release it.
- Can make user level / spinning semaphores
  - Same problem as with locks etc
- More useful if the waiting process is blocked and put on a queue (atomically) with the help of the operating system.

Pthreads semaphores

```
#include <semaphore.h>

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

int sem_post(sem_t *sem);

int sem_wait(sem_t *sem);

int sem_trywait(sem_t *sem);


int sem_timedwait(sem_t *sem,
       const struct timespec *abs_timeout);
```

From manpage of sem_init, sem_post, sem_wait

Semaphores

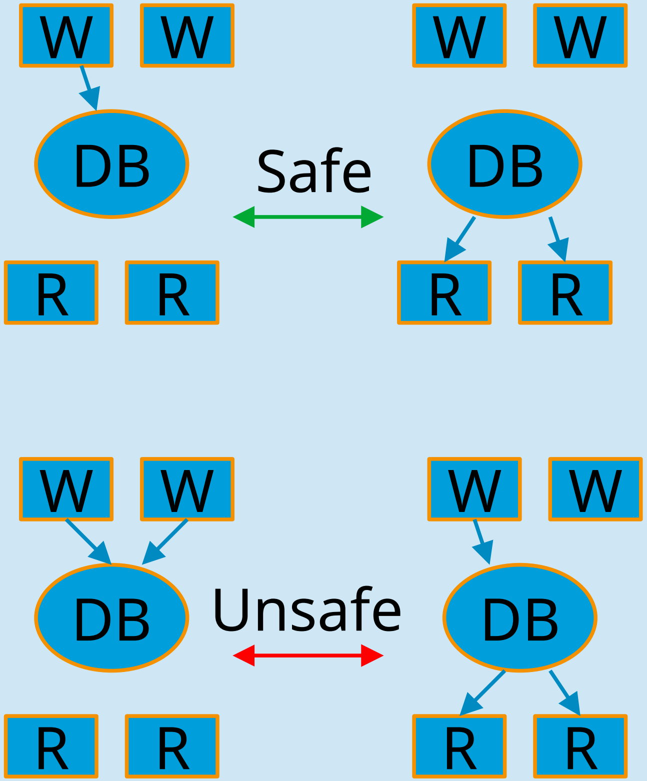
- Solving producer-consumer using semaphores
- To understand how it works:
  - The mutex semaphore is used to protect the queue datastructure
  - Note empty=N and full=N
  - Try looking at two cases first
    1) Producer running until the queue is full
    2) Consumer running until it blocks

```
#define N 100                        /* number of slots in the buffer */
typedef int semaphore;               /* semaphores are a special kind of int */
semaphore mutex = 1;                 /* controls access to critical region */
semaphore empty = N;                 /* counts empty buffer slots */
semaphore full = 0;                  /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                  /* TRUE is the constant 1 */
          item = produce_item();     /* generate something to put in buffer */
          down(&empty);              /* decrement empty count */
          down(&mutex);              /* enter critical region */
          insert_item(item);         /* put new item in buffer */
          up(&mutex);                /* leave critical region */
          up(&full);                 /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                  /* infinite loop */
          down(&full);               /* decrement full count */
          down(&mutex);              /* enter critical region */
          item = remove_item();      /* take item from buffer */
          up(&mutex);                /* leave critical region */
          up(&empty);                /* increment count of empty slots */
          consume_item(item);        /* do something with the item */
     }
}
```

Figure 2-28. The producer-consumer problem using semaphores.
Modern Operating Systems, Tanenbaum & Bos

Readers and writers problem

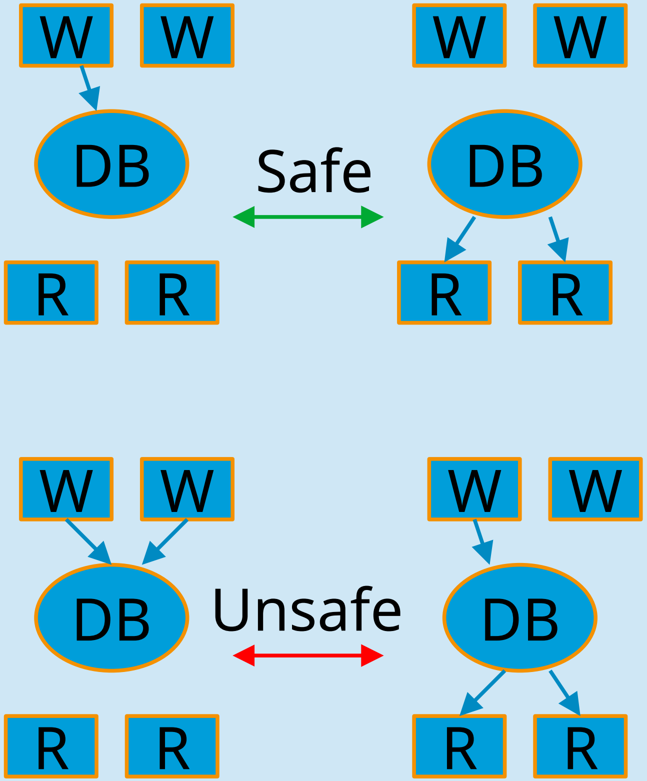
- 1+ readers trying to read state
- 1+ writers trying to write state
- Special cases where #r or #w is 1.
- Rules:
  - If no writers are active, then multiple readers can be active at the same time (no changes to the state)
  - If a writer is changing state, then we should not allow any readers (observe inconsistent state) or any other writers (inconsistent updates to state)
- Can get better performance as multiple readers can be serviced at the same time.
- Example from the book: airline reservation database.



Readers and writers problem

Need to keep track of the following invariants

- Number of readers (nr):
  - 0..R if nw == 0, 0 if nw > 0
- Number of writers:
  - 0..1if nr == 0, 0 if nr > 0



One solution using semaphores

Note the asymmetry

- Writer(s) lock the entire db so only one writer can be in at the time, and it also locks out readers
- The first reader locks down the database on behalf of other readers
  - Releases the mutex semaphore to let other readers in
  - The last reader to exit releases the db
- Note: there is a fairness issue with this solution. A continuous stream of readers may keep the writers locked out indefinitely.

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
      while (TRUE) {                    /* repeat forever */
            down(&mutex);               /* get exclusive access to rc */
            rc = rc + 1;                /* one reader more now */
            if (rc == 1) down(&db);     /* if this is the first reader ... */
            up(&mutex);                 /* release exclusive access to rc */
            read_data_base();           /* access the data */
            down(&mutex);               /* get exclusive access to rc */
            rc = rc − 1;                /* one reader fewer now */
            if (rc == 0) up(&db);       /* if this is the last reader ... */
            up(&mutex);                 /* release exclusive access to rc */
            use_data_read();            /* noncritical region */
      }
}

void writer(void)
{
      while (TRUE) {                    /* repeat forever */
            think_up_data();            /* noncritical region */
            down(&db);                  /* get exclusive access */
            write_data_base();          /* update the data */
            up(&db);                    /* release exclusive access */
      }
}
```

Figure 2-29. A solution to the readers and writers problem.
Modern Operating Systems, Tanenbaum & Bos