Betriebssysteme

08. Practical Synchronization by Example

Prof. Dr.-Ing. Frank Bellosa | WT 2016/2017

KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT) – OPERATING SYSTEMS GROUP

Where we ended last lecture

- There is often the need for processes or threads to communicate
  - Message passing facilities provide explicit send and receive functions to exchange messages
  - Implicitly shared memory between threads or explicitly shared memory between processes allows exchanging information by modifying shared state
- When communicating, data races need to be taken into account
- Common techniques to synchronizes access to shared data include
  - Interlocked atomic operations
  - Spinlocks
  - Semaphores
  - Futexes

Classic Synchronization Problems

POSIX Thread Synchronization

- POSIX provides a number of synchronization constructs that are based on spinlocks and semaphores described in the last lecture
- `pthread_mutex_t` provides the functionality of the previously discussed binary semaphore
  - Implemented as a futex in Linux
- `pthread_cond_t` implement condition variables which can be used in scenarios in which a counting semaphore is needed albeit with easier usage semantics
- `pthread_rwlock_t` implements reader-writer-locks in POSIX

Pthread Mutex

Pthread Mutex call	Description
`pthread_mutex_init`	Create and initialize a new mutex
`pthread_mutex_destroy`	Destroy and free existing mutex
`pthread_mutex_lock`	Enter critical section or block
`pthread_mutex_trylock`	Enter critical section or return with error
`pthread_mutex_unlock`	Leave critical section

- Statically allocated mutexes cannot be initialized with **`pthread_mutex_init`**
  - Initialize such mutexes with the **`PTHREAD_MUTEX_INITIALIZER`** constant
- Mutexes that are allocated on the heap with **`malloc`** need to be destroyed with **`pthread_mutex_destroy`** before freeing them
- **`pthread_mutex_trylock`** returns **`EBUSY`** if it cannot enter the CS

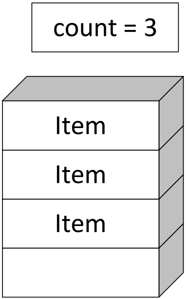
Pthread Mutex Example

```c
typedef struct {
   int count;
   pthread_mutex_t lock;
} Count;

void inc( Count *num )
{
   pthread_mutex_lock( &num.lock );
   num.count++;
   pthread_mutex_unlock( &num.lock );
}

void dec( Count *num )
{
   pthread_mutex_lock( &num.lock );
   num.count++;
   pthread_mutex_unlock( &num.lock );
}
```

```c
int main()
{
  Count num;
  num.count = 0;
  pthread_mutex_init(
     &num.lock, NULL );

  int i;
  #pragma omp parallel for
  for( i = 0; i < 42; ++i )
  {
     inc( &num );
     dec( &num );
  }
  [...]
  pthread_mutex_destroy(
    &num.lock );
  [...]
```

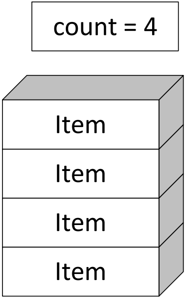
Producer-Consumer Problem

- Consider the producer-consumer problem (also known as bounded-buffer problem)
  - A buffer is shared between a producer and a consumer (here: LIFO)
  - An integer **`count`** keeps track of the number of currently available (previously produced) items
  - Every time, the producer produces an item, it places it in the buffer and increments **`count`**
  - When the buffer is full, the producer needs to sleep until the consumer consumed an item
  - When the consumer consumes an item, it removes the item from the buffer and decrements **`count`**
  - When the buffer is empty, the consumer needs to sleep until the producer produces an item



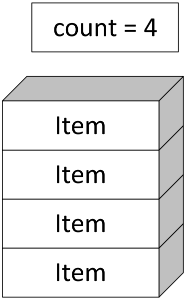
Producer-Consumer Problem

- Consider the producer-consumer problem (also known as bounded-buffer problem)
  - A buffer is shared between a producer and a consumer (here: LIFO)
  - An integer **`count`** keeps track of the number of currently available (previously produced) items
  - Every time, the producer produces an item, it places it in the buffer and increments **`count`**
  - When the buffer is full, the producer needs to sleep until the consumer consumed an item
  - When the consumer consumes an item, it removes the item from the buffer and decrements **`count`**
  - When the buffer is empty, the consumer needs to sleep until the producer produces an item



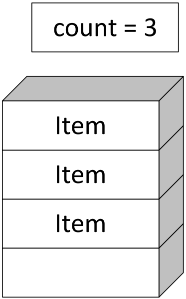
Producer-Consumer Problem

- Consider the producer-consumer problem (also known as bounded-buffer problem)
  - A buffer is shared between a producer and a consumer (here: LIFO)
  - An integer **`count`** keeps track of the number of currently available (previously produced) items
  - Every time, the producer produces an item, it places it in the buffer and increments **`count`**
  - When the buffer is full, the producer needs to sleep until the consumer consumed an item
  - When the consumer consumes an item, it removes the item from the buffer and decrements **`count`**
  - When the buffer is empty, the consumer needs to sleep until the producer produces an item



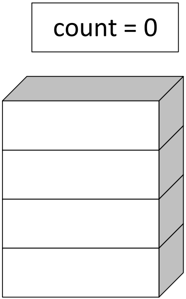
Producer-Consumer Problem

- Consider the producer-consumer problem (also known as bounded-buffer problem)
  - A buffer is shared between a producer and a consumer (here: LIFO)
  - An integer **`count`** keeps track of the number of currently available (previously produced) items
  - Every time, the producer produces an item, it places it in the buffer and increments **`count`**
  - When the buffer is full, the producer needs to sleep until the consumer consumed an item
  - When the consumer consumes an item, it removes the item from the buffer and decrements **`count`**
  - When the buffer is empty, the consumer needs to sleep until the producer produces an item



Producer-Consumer Problem

- Consider the producer-consumer problem (also known as bounded-buffer problem)
  - A buffer is shared between a producer and a consumer (here: LIFO)
  - An integer **`count`** keeps track of the number of currently available (previously produced) items
  - Every time, the producer produces an item, it places it in the buffer and increments **`count`**
  - When the buffer is full, the producer needs to sleep until the consumer consumed an item
  - When the consumer consumes an item, it removes the item from the buffer and decrements **`count`**
  - When the buffer is empty, the consumer needs to sleep until the producer produces an item



Producer-Consumer Problem

```
void producer()
{
   Item newItem;

   for(;;) // ever
   {
      newItem = produce();

      if( count == MAX_ITEMS )
         sleep();

      insert( newItem );
      count++;

      if( count == 1 )
         wake_up( consumer );

   }
}
```

```
void consumer()
{
   Item item;

   for(;;) // ever
   {
      if( count == 0 )
         sleep();

      item = remove();
      count--;

      if( count == MAX_ITEMS - 1 )
         wake_up( producer );

      consume( item );

   }
}
```

Producer-Consumer Problem

```
void producer()
{
   Item newItem;

   for(;;) // ever
   {
      newItem = produce();

      if( count == MAX_ITEMS )
         sleep();

      insert( newItem );
      count++;

      if( count == 1 )
         wake_up( consumer );

   }
}
```

```
void consumer()
{
   Item item;

   for(;;) // ever
   {
      if( count == 0 )
         sleep();

      item = remove();
      count--;

      if( count == MAX_ITEMS - 1 )
         wake_up( producer );

      consume( item );

   }
}
```

- Race condition on **`count`** as demonstrated in last lecture

Non-Solution with mutex

```
void producer()
{
   Item newItem;

   for(;;) // ever
   {
      newItem = produce();

      if( count == MAX_ITEMS )
         sleep();
      mutex_lock( &lock );
      insert( newItem );
      count++;
      mutex_unlock( &lock );
      if( count == 1 )
         wake_up( consumer );

   }
}
```

```
void consumer()
{
   Item item;

   for(;;) // ever
   {
      if( count == 0 )
         sleep();
      mutex_lock( &lock );
      item = remove();
      count--;
      mutex_unlock( &lock );
      if( count == MAX_ITEMS - 1 )
         wake_up( producer );

      consume( item );

   }
}
```

Non-Solution with mutex

```
void producer()
{
   Item newItem;

   for(;;) // ever
   {
      newItem = produce();

      if( count == MAX_ITEMS )
         sleep();
      mutex_lock( &lock );
      insert( newItem );
      count++;
      mutex_unlock( &lock );
      if( count == 1 )
         wake_up( consumer );

   }
}
```

```
void consumer()
{
   Item item;

   for(;;) // ever
   {
      if( count == 0 )
         sleep();
      mutex_lock( &lock );
      item = remove();
      count--;
      mutex_unlock( &lock );
      if( count == MAX_ITEMS - 1 )
         wake_up( producer );

      consume( item );

   }
}
```

- **`if`** statements can still be racy

Another non-Solution with mutex

```
void producer()
{
   Item newItem;

   for(;;) // ever
   {
      newItem = produce();
      mutex_lock( &lock );
      if( count == MAX_ITEMS )
         sleep();

      insert( newItem );
      count++;

      if( count == 1 )
         wake_up( consumer );
      mutex_unlock( &lock );
   }
}
```

```
void consumer()
{
   Item item;

   for(;;) // ever
   {  mutex_lock( &lock );
      if( count == 0 )
         sleep();

      item = remove();
      count--;

      if( count == MAX_ITEMS - 1 )
         wake_up( producer );
      mutex_unlock( &lock );
      consume( item );

   }
}
```

Another non-Solution with mutex

```
void producer()
{
   Item newItem;

   for(;;) // ever
   {
      newItem = produce();
      mutex_lock( &lock );
      if( count == MAX_ITEMS )
         sleep();

      insert( newItem );
      count++;

      if( count == 1 )
         wake_up( consumer );
      mutex_unlock( &lock );
   }
}
```

```
void consumer()
{
   Item item;

   for(;;) // ever
   {  mutex_lock( &lock );
      if( count == 0 )
         sleep();

      item = remove();
      count--;

      if( count == MAX_ITEMS - 1 )
         wake_up( producer );
      mutex_unlock( &lock );
      consume( item );

   }
}
```

- One cannot work while the other sleeps with lock held (deadlock)

Final non-Solution with mutex

```
void producer()
{   [...]
    for(;;) // ever
    {
        newItem = produce();
        mutex_lock( &lock );
        if( count == MAX_ITEMS )
        {
            mutex_unlock( &lock );
            sleep();
            mutex_lock( &lock );
        }
        insert( newItem );
        count++;

        if( count == 1 )
            wake_up( consumer );
        mutex_unlock( &lock );
    }
}
```

```
void consumer()
{   [...]
    for(;;) // ever
    {   mutex_lock( &lock );
        if( count == 0 )
        {
            mutex_unlock( &lock );
            sleep();
            mutex_lock( &lock );
        }

        item = remove();
        count--;

        if( count == MAX_ITEMS - 1 )
            wake_up( producer );
        mutex_unlock( &lock );
        consume( item );
    }
}
```

Final non-Solution with mutex

```
void producer()
{   [...]
    for(;;) // ever
    {
        newItem = produce();
        mutex_lock( &lock );
        if( count == MAX_ITEMS )
        {
            mutex_unlock( &lock );
            sleep();
            mutex_lock( &lock );
        }
        insert( newItem );
        count++;

        if( count == 1 )
            wake_up( consumer );
        mutex_unlock( &lock );
    }
}
```

```
void consumer()
{   [...]
    for(;;) // ever
    {   mutex_lock( &lock );
        if( count == 0 )
        {
            mutex_unlock( &lock );
            sleep();
            mutex_lock( &lock );
        }

        item = remove();
        count--;

        if( count == MAX_ITEMS - 1 )
            wake_up( producer );
        mutex_unlock( &lock );
        consume( item );
    }
}
```

- Still racy and can cause signal loss

Condition Variables

- Problem can be solved with a mutex and 2 counting semaphores
  - Hard to understand
  - Hard to get right
  - Hard to transfer to other problems

- Condition Variables (CV) allow blocking until a condition is met

- Condition variables are usually suitable for the same problems but they are much easier to “get right”

- Idea:
  - New operation that performs unlock, sleep, lock atomically
  - New wake-up operation that is called with lock held

  → Simple mutex lock/unlock around CS + no signal loss

Pthread Condition Variables

Pthread CV call	Description
`pthread_cond_init`	Create and initialize a new CV
`pthread_cond_destroy`	Destroy and free an existing CV
`pthread_cond_wait`	Block waiting for a signal
`pthread_cond_timedwait`	Block waiting for a signal or timer
`pthread_cond_signal`	Signal another thread to wake up
`pthread_cond_broadcast`	Signal all threads to wake up

Solution with Condition Variables

- Two condition variables: **`more`** and **`less`**

```
void producer()
{
  Item newItem;

  for(;;) // ever
  {
    newItem = produce();

    mutex_lock( &lock );
    while( count == MAX_ITEMS )
      cond_wait( &less, &lock );

    insert( newItem );
    count++;

    cond_signal( &more );
    mutex_unlock( &lock );
  }
}
```

```
void consumer()
{
  Item item;

  for(;;) // ever
  {
    mutex_lock( &lock );
    while( count == 0 )
      cond_wait( &more, &lock );

    item = remove();
    count--;

    cond_signal( &less );
    mutex_unlock( &lock );

    consume( item );
  }
}
```

Readers-Writers Problem

- Problem: Model access to shared data structures
  - Many threads compete to read or write the same data
  - Readers only read the data set; they do not perform any updates
  - Writers can both read and write

- Using a single mutex for read and write operations is not a good solution, as it unnecessarily blocks out multiple readers while no writer is present

- Idea: Locking should reflect different semantics for reading data and for writing data
  - If no thread writes, multiple readers may be present
  - If a thread writes, no other readers and writers are allowed

1^{st} Readers-Writers Problem: Readers Preference

- No reader should have to wait if other readers are already present

```
void writer()
{
   for(;;) // ever
   {
      // generate data to write

      wait( write_lock );

      // write data

      signal( write_lock );
   }
}
```

```
void reader()
{
   for(;;) // ever
   {
      mutex_lock( &rc_lock );
      readerscount++ ;
      if( readerscount == 1 )
         wait( &write_lock );
      mutex_unlock( &rc_lock );

      // read data

      mutex_lock( &rc_lock );
      readerscount--;
      if (readerscount == 0)
         signal( &write_lock );
      mutex_unlock( &rc_lock );
   }
}
```

- Writers cannot acquire **`write_lock`** until the last reader leaves the critical section

2^{nd} Readers-Writers Problem: Writers Preference

- No writer shall be kept waiting longer than absolutely necessary
- Code is analogous to 1^{st} readers-writers problem but with separate readers- and writers-counts
- Read “Concurrent Control with Readers and Writers” by Randell if you are interested in code for a solution
- 1^{st} and 2^{nd} readers-writers problem have the same issue:
  - Readers preference → writers can starve
  - Writers preference → readers can starve

3^{rd} Readers-Writers Problem: Bounded Waiting

- No thread shall starve
- POSIX threads contains readers-writers locks to address this issue

Pthread Mutex call	Description
`pthread_rwlock_init`	Create and initialize a new RW lock
`pthread_rwlock_destroy`	Destroy and free an existing RW lock
`pthread_rwlock_rdlock`	Block until reader lock acquired
`pthread_rwlock_wrlock`	Block until writer lock acquired
`pthread_rwlock_unlock`	Leave critical section

- Multiple readers but only a single writer are let into the CS
- If readers are present while a writer tries to enter the CS then
  - don't let further readers in
  - block until readers finish
  - let writer in

POSIX Readers-Writers Locks

- Readers-writers locks make solving the 3*rd* readers-writers problem a non-issue…

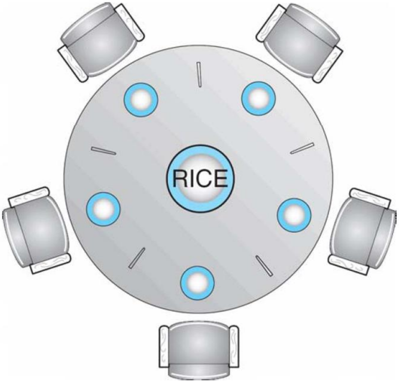
```
void writer()
{
   for(;;) // ever
   {
      rwlock_wrlock( rw_lock );

      // write data

      rwlock_unlock( rw_lock );
   }
}
```

```
void reader()
{
   for(;;) // ever
   {
      rwlock_rdlock( rw_lock );

      // read data

      rwlock_unlock( rw_lock );
   }
}
```

- …unless you have to implement the readers-writers locks

Dining-Philosophers Problem

- Cyclic workflow of 5 philosophers
  1. Think
  2. Get hungry
  3. Grab for one chopstick
  4. Grab for other chopstick
  5. Eat
  6. Put down chopsticks



- Ground rules
  - No communication
  - No “atomic” grabbing of both chopsticks
  - No wrestling

- Models threads competing for limited number of resources (e.g., I/O devices)

Dining-Philosophers Problem

- Naïve solution with **`mutex_t chopstick[5]`** representing the chopsticks
  - What happens if all philosophers grab their left chopstick at once?

```
void philosopher( int i )
{
   for(;;) // ever
   {
      mutex_lock( chopstick[i] );
      mutex_lock( chopstick[(i + 1) % 5] );
      //  eat
      mutex_unlock( chopstick[i] );
      mutex_unlock( chopstick[(i + 1) % 5] );
      //  think
   }
}
```

- Deadlock workarounds
  - Just 4 philosophers allowed at a table of 5 (example for deadlock avoidance)
  - Odd philosophers take left chopstick first, even philosophers take right chopstick first (example for deadlock prevention)

Deadlocks

Deadlock Conditions

Deadlocks can arise if all four conditions hold simultaneously:

1. Mutual exclusion
   - Limited access to resource
   - Resource can only be shared with a finite amount of users
2. Hold and wait
   - Wait for next resource while already holding at least one
3. No preemption
   - Once the resource is granted, it cannot be taken away but only handed back voluntarily
4. Circular wait
   - Possibility of circularity in graph of requests

Example: Deadlock Conditions

1. Only one intersection
2. Cars block part of the intersection while waiting for the rest
3. Cars don’t diminish into thin air
4. Every one of the four sides waits for the cars that come from the right to give way

Deadlock countermeasures

Three approaches to dealing with deadlocks:

- Prevention
  Pro-active, make deadlocks impossible to occur
- Avoidance
  Decide on allowed actions based on a-priori knowledge
- Detection
  React after deadlock happened (recovery)

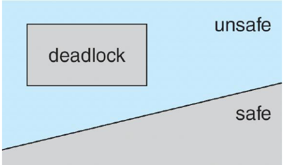
Deadlock Prevention

Negate at least one of the required deadlock conditions:

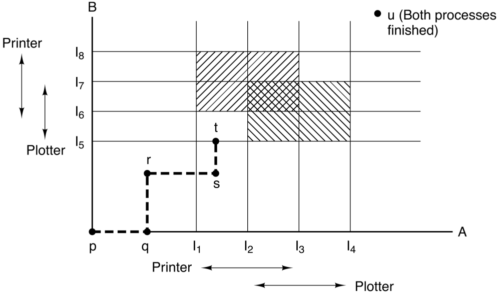
1. Mutual exclusion
   - Buy more resources, split into pieces, virtualize $\rightarrow$ "infinite" # of instances
2. Hold and wait
   - Get all resources en-bloque
   - 2-phase-locking
3. No preemption
   - Virtualize to make preemptable
     - virtual vs. physical memory
     - spooling (printer)
4. Circular wait
   - Ordering of resources
   - Prevent deadlocks with partial order on resources!
     - E.g., always acquire mutex m_1 before m_2

Deadlock Avoidance

- If a system is in safe state
  → no deadlocks
- If a system is in unsafe state
  → deadlocks possible

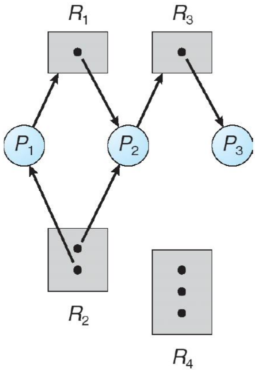


- Deadlock Avoidance
  On every resource request: decide if system stays in safe state
  - Needs a-priori information (e.g., max resources needed)
  - Resource Allocation Graph



Resource Allocation Graph (RAG)

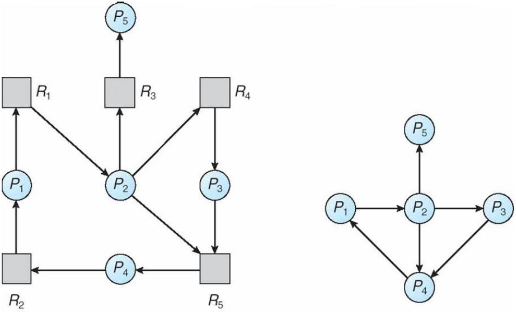
- View system state as graph
  - Processes are round nodes
  - Resources are square nodes
- Every instance of a resource is depicted as a dot in the resource node
- Resource requests and assignments are edges
  - Resource pointing to process means:
    Resource is assigned to processes
  - Process pointing to resource means:
    Process is requesting resource
  - Process *may* request resource:
    Claim edge, depicted as dotted line
    (without claim edges only one
    point in time depictable)



Deadlock Detection

Allow system to enter deadlock → detection → recovery scheme

- Maintain Wait-For Graph (WFG)
  - Nodes are processes
  - Edge represents “wait for” relationship (Like RAG, but without resources)



- Periodically invoke an algorithm that searches for a cycle in the graph
  - If there is a cycle, there exists a deadlock

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim
  - Minimize cost

- Rollback
  - Perform periodic snapshots
  - Abort process to preempt resources

  → Restart process from last safe state

- Starvation
  - Same process may always be picked as victim

  → Include number of rollbacks in cost factor

Summary

- Classical synchronization problems model synchronization problems that occur in reality
  - Producer-Consumer Problem: Shared use of buffers/queues
  - Readers-Writers Problem: Shared access to data structures
  - Dining Philosophers: Competition for limited resources
- Such synchronization problems occur very often when programming operating systems
- The parallelism introduced by multiple processors and the concurrency introduced by multiprogramming needs to be considered carefully when writing an OS
- Poorly synchronized code can lead to starvation, priority inversion, or deadlocks

Further Reading

- Tanenbaum/Bos, “Modern Operating Systems”, 4th Edition:
  - Pages 119–148
  - Pages 167–173
  - Chapter 6
- Stevens, Rago: Advanced Programing in the UNIX Environment:
  - Pages 367–386