Synchronization and Deadlocks

1DV512 – Operating Systems

Dr. Kostiantyn Kucher

kostiantyn.kucher@lnu.se

November 17, 2020

Based on the Operating System Concepts slides by Silberschatz, Galvin, and Gagne (2018)

Suggested OSC book complement: Chapters (3 &) 6 & 7

Agenda

- Motivation and Introduction
- Inter-Process Communication
- Process and Thread Synchronization Approaches
- Deadlocks
- Summary

Motivation

- We have previously established the concepts of a *process* and a *thread*
- We have also discussed how such tasks can be *scheduled* for execution to proceed with their computations

- What if the computations depend on the results or intermediate data from another process or thread?
  - We discussed how a parent process can check the exit status of the child process, but there is surely more to this topic!
  - *A lot* of processes and threads can be executed concurrently by the OS $\Rightarrow$ concurrent access to shared data may result in data inconsistency
- We must discuss the models for exchanging data between the tasks, and preventing problematic situations that might arise

Agenda

- Motivation and Introduction
- Inter-Process Communication
- Process and Thread Synchronization Approaches
- Deadlocks
- Summary

Main Models of Inter-Process Communication

- Processes within a system may be *independent* or *cooperating*
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity and convenience
- Cooperating processes need *interprocess communication (IPC)*

- Two main models of IPC:
  - *Shared memory* ⇒ faster for common access to larger amounts of data
  - *Message passing* ⇒ easier and more efficient for smaller amounts of data
- Major issue: how to provide a mechanism that will allow the user processes to *synchronize*

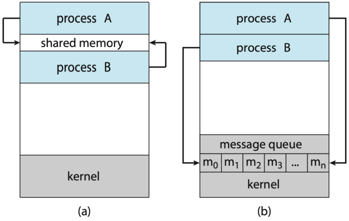

Fig. 3.11 in OSC book

Producer-Consumer Problem

- Common paradigm for cooperating processes ⇒ *producer* process produces information that is consumed by a *consumer* process
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced
- One possible solution for the producer-consumer problem ⇒ shared memory
- The communication is under the control of the user processes, not the operating system
- A buffer of items that can be filled by the producer and emptied by the consumer, with two main variations:
  - *unbounded-buffer* places no practical limit on the size of the buffer
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - *bounded-buffer* assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume

Producer-Consumer Problem (cont.)

- The shared `buffer` is implemented as a circular array with two logical pointers: `in` and `out`
  - `in` points to the next free position in the buffer
  - `out` points to the first full position in the buffer

```
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Sec. 3.5 in OSC book

- The buffer is full when `((in + 1) % BUFFER SIZE)` == `out`, and empty when `in` == **out**
- Simpler solution allows the usage of `BUFFER_SIZE`−1 elements
- Solution using all `BUFFER_SIZE` elements can lead to *race conditions* if both processes try to access the buffer simultaneously ⇒ more on synchronization later!

```
item next_produced;

while (true) {
     /* produce an item in next_produced */

     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Fig. 3.12 in OSC book

```
item next_consumed;

while (true) {
     while (in == out)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;

     /* consume the item in next_consumed */
}
```

Fig. 3.13 in OSC book

Message Passing

- Message passing is an alternative to shared memory ⇒ support for `send(message)` and `receive(message)` operations, with either fixed or variable message size
- *Direct* communication ⇒ a pair of processes name each other explicitly and send messages to each other (can also be unidirectional)
- *Indirect* communication ⇒ messages are directed and received from mailboxes (also referred to as *ports*)
- Message passing may be either *blocking* (synchronous) or *non-blocking* (asynchronous)
- Messages reside in a temporary buffering queue of *zero* capacity (⇒ sender must wait for receiver), *bounded*/finite capacity (⇒ sender must wait if full queue), or *unbounded* capacity (⇒ sender never waits)
- The producer-consumer problem can be approached with message passing, too:

```
message next_produced;

while (true) {
     /* produce an item in next_produced */

     send(next_produced);
}
```

Fig. 3.15 in OSC book

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

Fig. 3.15 in OSC book

IPC Examples

- Shared memory
- *Pipes*
  - Once established, reading and writing operations similar to regular file read/write API
  - Can be typically uni- or bidirectional
  - Unidirectional (*ordinary* or *anonymous*) pipes ⇒ a pair of pipes might be necessary to communicate between a parent and a child process
  - Biredictional (*named*) pipes ⇒ often visible as special files in the file system, can be used by multiple processes
- *Sockets*
  - Communication between multiple *hosts* (⇒ *network sockets*) or as an efficient IPC tool within the same host (⇒ *Unix domain sockets*)
- *Remote Procedure Calls (RPC)*

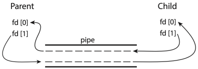

Fig. 3.20 in OSC book

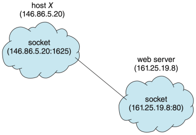

Fig. 3.26 in OSC book

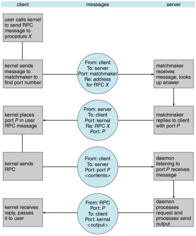

Fig. 3.29 in OSC book

Agenda

- Motivation and Introduction
- Inter-Process Communication
- Process and Thread Synchronization Approaches
- Deadlocks
- Summary

Synchronization and The Critical Section Problem

- Previously, we saw how concurrent access and manipulation of shared data in the consumer-producer problem can lead to inconsistency issues ⇒ *race conditions*
- One scenario: several processes invoking `fork()` at the *same* time and receiving the same child PID!
- *Synchronization* issues even more important, given the prominence of modern multicore systems with support for multithreading

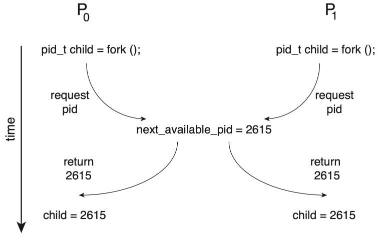

Fig. 6.2 in OSC book

Critical Section Problem

- Critical section ⇒ a section of code where the data shared with another process is accessed or updated
- While one process is in the critical section ⇒ no other processes allowed to enter their critical section!
- Each process must ask permission to enter critical section in *entry section*, may follow critical section with *exit section*, then *remainder section*
- Requirements for possible solution designs:
  - *Mutual exclusion*
  - *Progress* ⇒ if several processes are waiting to enter the critical section, one of them should eventually be able to do it!
  - *Bounded waiting*
- The problem is very important for kernel-mode code
- Solutions can be completely software-based (⇒ often not sufficient for multicore systems) or might involve support from hardware (⇒ *memory barriers/fences*, *atomic instructions* such as *test and set* or *compare and swap*, and *atomic variables*)

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

Fig. 6.1 in OSC book

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Fig. 6.5 in OSC book

```
int compare_and_swap(int *value, int expected, int new_value) {
  int temp = *value;

  if (*value == expected)
    *value = new_value;

  return temp;
}
```

Fig. 6.7 in OSC book

Mutex Lock

- OS designers typically build software tools to solve critical section problem
- Simplest is *mutex lock* (using the mutual exclusion concept) ⇒ boolean variable indicating if lock is available or not
- Provides `acquire()` and `release()` operations ⇒ must be atomic! (usually implemented using atomic hardware instructions)
- If a process is forced to loop (*spin*) a lot while waiting ⇒ such mutex is also called a *spinlock*
  - Can be preferable, if the waiting time is shorter than the combined time of context switches!

```
while (true) {

    acquire lock

        critical section

    release lock

        remainder section

}
```

Fig. 6.10 in OSC book

```
do {
   while (test_and_set(&lock))
     ; /* do nothing */

     /* critical section */

   lock = false;

     /* remainder section */
} while (true);
```

Fig. 6.6 in OSC book

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

Fig. 6.8 in OSC book

Semaphore

- *Semaphore* ⇒ synchronization tool that provides more sophisticated ways (than mutex locks) for processes to synchronize their activities
- Relies on an integer counter value `S` that can only be accessed using two atomic operations, `wait()` and `signal()`
- *Binary* semaphore ⇒ `S` is either 0 or 1 ⇒ equivalent to a mutex lock
- *Counting* semaphore ⇒ `S` has an unrestricted range
- Powerful tool, but no guarantees that the users' code will use it correctly!

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

Sec. 6.6 in OSC book

```
signal(S) {
    S++;
}
```

Sec. 6.6 in OSC book

S_1;
`signal(synch);`

Sec. 6.6.1 in OSC book

`wait(synch);`
S_2;

Sec. 6.6.1 in OSC book

Monitor

- *Monitor* $\Rightarrow$ an high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

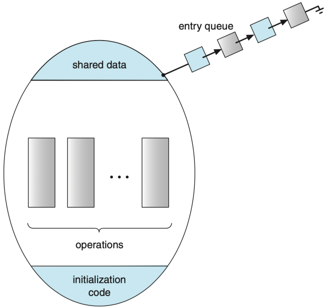

Fig. 6.12 in OSC book

Agenda

- Motivation and Introduction
- Inter-Process Communication
- Process and Thread Synchronization Approaches
- Deadlocks
- Summary

Deadlocks

- A system consists of a finite number of resources to be distributed among a number of competing processes or threads
- Each process normally utilizes a resource as follows:
    - request
    - use
    - release
- *Deadlock* ⇒ every task is waiting for an event that can be caused only by another task

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}
```

```
/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

Fig. 8.01 in OSC book

Deadlock Conditions

- Deadlock can arise if four conditions hold simultaneously:
  - Mutual exclusion: only one process at a time can use a resource
  - Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
  - No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - Circular wait: there is a cycle of processes waiting for each other's actions

Deadlock Handling Approaches

- Ignore the problem and pretend that deadlocks never occur in the system
- Ensure that the system will never enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover

Agenda

Summary

- The typical models for inter-process communication involve shared memory and message passing $\Rightarrow$ synchronization issues often exist
- Support for synchronization between processes and threads may involve software and hardware aspects
- The most typical approaches for synchronization include mutex locks and semaphores
- Deadlock situations can emerge based on the patterns of access to the resources

- **Additional demos** (source code + slides) on thread programming with Java will be published on MyMoodle $\Rightarrow$ incl. thread synchronization approaches