Operating systems
Problem set II
Uppsala University – Spring 2016

Linnea Ingmar
Elsa Rick

15 februari 2016

1 Threads vs processes

1. *What is the difference between processes and threads?*

   Multiple threads can exist within the same process. Threads within the same process can share resources such as memory and instructions (executable code). Processes do not share memory or instructions.

2. *Why is it more expensiveto create a new process compared to create a new thread?*

   It is more time consuming and takes more resources to create a new process compared to create a new thread, since threads share resources with each other.

2 Random numbers

1. *What happens when two or more threads call rand() and how is this different compared problem set 1 where we used rand() together with fork()?*

   From the Linux manual page:
   "The function rand() is not reentrant or thread-safe, since it uses hidden state that is modified on each call."
   The threads share the memory where the state of rand() is stored between each call. A call to rand() is not atomic, so there might be undefined behaviour when using rand() in a multi-threaded program. The threads might receive different sequences of random numbers, or they might get the same depending on the squeduling of read/write to the seed value. This is diffent to the behaviour of rand() in a multi-process program, where the sequences always will be the same if they have the same initial seed, since memory is not shared between the processes.

2. *What is the difference between rand() and rand_r()?*

   The rand_r() function is a reentrant function that takes a pointer to an unsigned int that is used to store states between calls. This can be used to ensure thread-safety, by passing different pointers to rand_r() in different threads. In that way the threads will not share the memory of the seed and the behaviour will be simular to a multi-process program using rand().

3 Need for synchronization

1. *Why do concurrent access to shared memory need to be synchronized?*

   To be able to guarantee the behaviour of a program, we need to synchronize the access to shared memory, so that a program does not behave differently on different calls.

2. *What is meant by a race condition (aka data race)?*

   A race condition is the behaviour of a program (or other system) where the output is dependent on the sequence of timing of other uncontrollable events. A data race is (which is *not* the same as a race condition according to the slides!) occurs when two or more intstructions from different threads, where at least one of these accesses is a write, access the same memory location without synchronization. Note that many race conditions are due to data race and that many data races lead to race conditions, but we can have race conditions without a data race, and all data races don't lead to race conditions!

4 Software based synchronization

1. *What are the problems with Petersson's solution?*

   It only works for two concurrent tasks.
   Software-based solutions such as Peterson's algorithm is not guaranteed to work on modern computer architectures due to reordering of memory operations.

5 Hardware support for synchronization

1. *Name two atomic instructions that can be used to implement mutex locks.*

   TestAndSet and Swap.

2. *What is meant by busy waiting?*

   Busy waiting is when a thread or process remains active, for example spinning in a while loop, spending CPU cycles, until a certain condition is fulfilled.

3. *How can spin locks be constructed using the atomic instructions from above?*

   This is how TestAndSet can be used:

```
do {
  while (TestAndSet(&lock))
  ; // busy wait

  // critical section

  lock = false;

  // non-critical section
} while(true);
```

   This is how Swap can be used:

```
do {
  while (key == true)
    Swap(&lock, &key); // busy wait

  // critical section

  lock = false;

  // non-critical section
} while(true);
```

6 Abstractions for synchronization

1. *What is a mutex lock?*

   It is a binary semaphore that provides mutual exclusion access to a critical section.

2. *What operations can be performed on a semaphore and how do these operations work?*

   init(): to initialize a semaphore
   destroy(): to destory (free memory?) for the semaphore
   wait(): to grab a lock if one is available, or wait for a lock to be available (wait for a cookie).
   signal(): to release the lock (put a cookie back to the jar).

3. *What is the difference between a mutex lock and a semaphore?*

   A mutex lock is a binary semaphore, which means that it can be implented using an integer that can take only the values 0 and 1.

4. *How can busy waiting be avoided?*

   By letting the semaphore have a waiting queue with blocked processes. A process that calls wait() when there is no lock available is blocked and put in the queue. Then the signal() call wakes up a process if there is a queue.

5. *How does the Java synchronized keyword work?*

   Each object in Java has associated with it a single lock. One can declare methods as synchronized. If a thread calls a method declared as `synchronized` on an object, that thread becomes the owner of the lock and all other threads that invoke synchronized methods on the same object are blocked until the first thread is done with the method on that object. One can also use synchronized statements to synchronize parts of methods. Then one must specify the object that provides the lock.

6. *How is this different compared to using semaphores?*

   `synchronized` allows exactly one thread to access the same method for a specific object. Semaphores allows up to **n** threads to access a specific code block. Also, you don't have to explicitly acquire and release the lock, it is done automatically when the thread enters and exits the method/code block.

7. *How is this different compared to using mutex locks?*

   That one has to explicitly acquire and release the lock. ???

8. *What is a monitor?*

   It is thread-safe class, object or module whose methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods.

9. *What problem with monitors is addressed by condition variables?*

   The problem that a thread waiting for a condition to be true would cause a deadlock in the monitor since the condition cannot be fulfilled until another thread can execute. A thread waiting for a condition variable is not considered to occupy the monitor, so another thread may enter the monitor.

10. *What is the difference between blocking (Hoare style) and non-blocking (Mesa style) conditi on variables?*

    Blocking condition variables (Hoare style) give priority to a signaled thread when signal() or wait() is called. Non-blocking condition variables (Mesa style) give priority to the signaling thread.

11. *What is a semaphore?*

    It is a synchronization tool to provide mutual exclusion.

7 Deadlock

1. *What is meant by a deadlock?*

   A deadlock occurs when two or more threads are waiting for an event to occur, that only one of the other waiting processes can perform.

2. *What are the four necessary conditions for deadlocks to occur?*

   From the book:

   (a) Mutual exclusion. At least one resource is held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

   (b) Hold and wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

   (c) No preemption. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

   (d) Circular wait. A set { P0 , P1 , ..., Pn } of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

   All four conditions must hold, but the circular-wait condition implies the hold-and-wait condition.

3. *What is the difference between deadlock prevention and deadlock avoidance?*

   Deadlock prevention is making sure at least one of the four necessary conditions do not hold, so that a deadlock cannot occur. Deadlock avoidance uses information about how resources are to be requested from each process, so that the system can decide whether or not a process should wait in order to avoid a possible future deadlock.

4. *Given a bank with numbered bank accounts, we have a function that transfers money from account A to account B. One way of writing the function is given below.*

```
  transfer (A, B, amount) // A and B are bank account numbers
{
    lock_account (A);
    lock_account (B);
    withdraw amount from account A;
    deposit amount into account B;
    unlock_account (B);
    unlock_account (A);
}
```

   One problem with this solution is that deadlocks can occur; for example, if two threads both do the following operations concurrently:

```
transfer (1, 2, 200);
transfer (2, 1, 100);
```

   In this case, a deadlock happens if each thread locks one account while waiting for the other thread to unlock the other account.

How can you change the transfer() function to prevent deadlocks?

By making the statements atomic? TODO

8 Starvation

1. *What is meant by starvation?*

   A process/thread is not given CPU for a very long time so that it cannot execute.

9 Bounded buffer

1. *How can semaphores be used to synchronize access to a bounded buffer?*

   One could use three semahores: one mutex provide mutual exclusion to the updates of nextp and nextc, one counting semaphore to keep count of data items in the buffer and one counting semaphore to keep count of the number of empty slots.

10 Dining philosophers

1. *When solving the Dining philosophers problem, how can:*

   (a) *Deadlocks be avoided? More than one solution?*
   By letting one philosopher pick up the right fork first, and everybody else pick up the left fork first.

   (b) *Starvation be avoided? More than one solution?*
   TODO