Sistemas Operativos: Concurrency (Part 3)

Pedro F. Souto (pfs@fe.up.pt)

March 30, 2012

Agenda

Classic Synchronization Problems

Synchronization in the Kernel

Additional Reading

Agenda

Classic Synchronization Problems

Synchronization in the Kernel

Additional Reading

Producer-Consumer Problem

- This is the name of the general problem behind the bounded-buffer problem.
  - The buffer instead of having a finite size is infinite
- Producers can always add an item to the “buffer”
  - Consumers still have to wait on empty “buffer”
- This is not as theoretical as it sounds
  - Implement the buffer with a linked-list rather than an array

```
typedef {
    void *head;
    void *tail;
} queue_t;
void enqueue(queue_t *q, void *obj);
void *dequeue(queue_t *q);
```

Sleeping Barber (1/2)

- Models the interaction between clients (customers) and servers (barber)

source: Modern Operating Systems, 2nd. Ed.

Sleeping Barber (2/2)

- The barber shop has *n* waiting seats, in addition to the barber chair
- The barber:
  - Can serve at most one customer at a time
  - If there are no customers, the barber sleeps (hence the name) in the barber chair
- When a customer arrives
  - If the barber is sleeping, it wakes it up
  - Otherwise, if there is an empty seat the customer waits
  - Else, it goes away
- Note that there is no data transfer in this problem as formulated
  - The issue is the synchronization between the processes
- A more general version of this problem considers the case where there is more than one barber.

```
void barber(void);
void customer(void);
```

The Readers and Writers Problem

- Models access to a “database”
  - This needs not be a relational database, but just a storage service supporting two basic operations

    `read()`
    `write()`
- Processes/threads can play two roles:

  readers which invoke only the `read()` operation
  writers which invoke the `write()` operation
- Correctness conditions:
  - A **writer** must access the database in mutual exclusion
    - I.e., when a writer accesses the database, no other process can access the database
    - In practice, we restrict access to a data item, while allowing concurrent accesses to different data items
  - **Readers** can access the database concurrently
- This has some similarities with the critical section problem
  - Mutual exclusion ensures correctness

The Readers and Writers Problem: Fairness (1/2)

The issue is ensure efficiency with fairness

When a reader is accessing the DB

- A writer trying to access the DB **must** wait
- A reader trying to access the DB may go ahead

  Issue How to prevent writers' starvation?
  - Must ensure that the arrival of "new" readers does not prevent a writer from ever accessing the DB

When a writer "leaves" the DB

- If there are:
  - Only writers waiting
  - Or, only readers waiting

  the action to take is obvious.

  Issue What if there are readers and writers wishing to access the DB?

The Readers and Writers Problem: Fairness (2/2)

- Solutions that may lead to starvation are more or less straightforward
  - And so are solutions that ensure mutual exclusion
- Solutions that ensure fairness efficiently are more difficult
- An approach is to try to be efficient without affecting fairness:
  - If a writer is made to wait, no further reads are allowed to access the database
  - When a writer “leaves” the database, acess is given to the process that has been waiting the longest.
    - If that process is a reader, all other readers wishing to access the DB are also allowed to do it
- Implementing this solution with available synchronization mechanisms is not always easy
  - Most often, there is no way to know which process has been waiting for longer
  - In this case, we may have to be less fair, but still avoid starvation

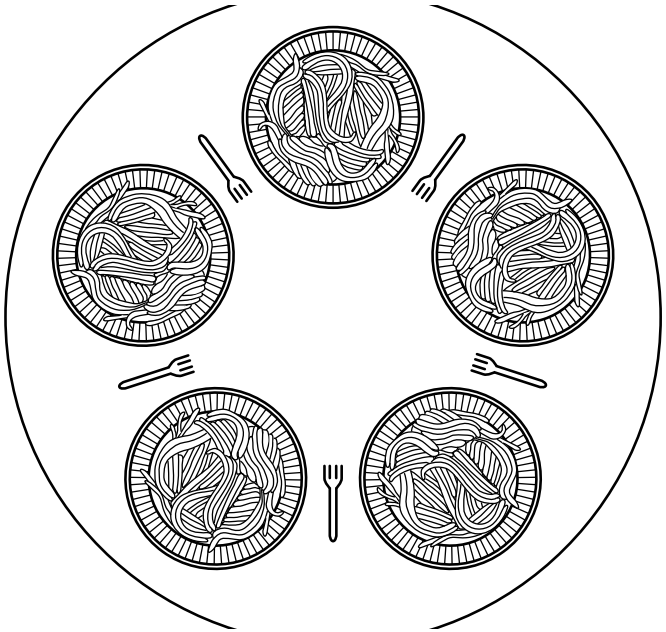
Read/Write Locks: POSIX API

```
// Life cycle calls
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

// For locking a read-write lock for reading
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

// For locking a read-write lock for writing
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);

// Timed lock calls: thread blocks at most abs_timeout
int pthread_rwlock_timedrdlock(pthread_rwlock_t *rwlock,
                const struct timespec *restrict abs_timeout);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *rwlock,
                const struct timespec *restrict abs_timeout);
```

- Apparently not available in Linux

Dining Philosophers

- Models the problem of allocating several resources among several processes without **deadlocks** nor **starvation**



src: Modern Operating Systems, 2nd. Ed.

- Five philosophers seat at a round table, with one plate of spaghetti per philosopher and one fork between each pair of plates
- The life of a philosopher consists of alternating periods of eating and thinking
- In order to eat, a philosopher needs to pick up both forks next to his plate
- After eating, a philosopher puts down both forks before starting to think

Dining Philosophers: First Try

Idea Each philosopher picks up first the fork on its left (assuming counterclockwise numbering)

```
#define N 5                    /* Number of philosophers */

void philosopher(int i) {
    while(1) {
        think();
        take_fork(i);
        take_fork((i+1)%N);
        eat();
        put_fork(i);
        put_fork((i+1)%N);
    }
}
```

Problem If all philosopher pick their left fork at the same time, none of them will be able to pick their right fork: deadlock

Dining Philosophers: More Tries

Solution (not really) If a philosopher cannot pick up its right fork, it puts down its left fork, and tries again later.

Problem If later all philosophers pick their left fork simultaneously, the same sequence of events can be replayed forever

- Although no process blocks, there is no progress, a situation known as **starvation**

- One possible solution is for each philosopher to pick up both its forks in mutual exclusion.
  - If they are not both available, it must pick none, and leave the critical section
- Yet another solution is to use asymmetry:
  - Odd numbered philosophers pick up first their left fork
  - Whereas even numbered philosophers pick up first their right fork

Agenda

Classic Synchronization Problems

Synchronization in the Kernel

Additional Reading

Facets of Kernel Synchronization

Implementation of the synchronization mechanisms

- Many synchronization mechanisms have to be implemented by the kernel

Synchronization in the kernel itself

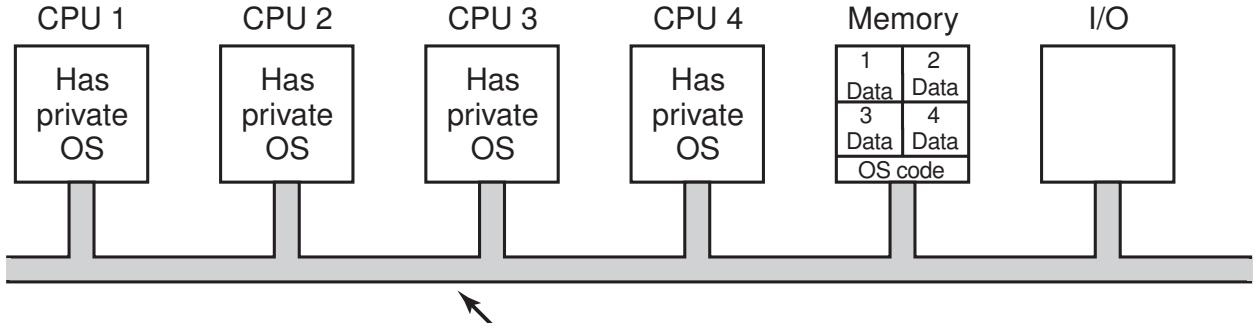
- Several processes/threads can make system calls concurrently
  - The kernel code implementing the system calls may modify kernel data structures
- Interrupt handlers need to access kernel data structures that may be accessed by other parts of the kernel code
- The concurrent execution of kernel code by different processes/threads and of interrupt handlers requires synchronization to prevent race conditions

Nonpreemptive Kernels

- Solution used mostly with uniprocessors
- A process/thread running in kernel mode is never preempted. A process/thread in kernel mode runs until it:
  - Exits the kernel (the system call returns)
  - Blocks
  - Voluntarily yields the CPU
- By careful programming, it is possible to avoid race conditions between processes/threads running in kernel mode
- Race conditions with interrupt handlers can be avoided by:
  - Disabling interrupts when accessing shared data structures
  - Often, the HW allows to selectively inhibiting interrupts
    - This allows the system to be more responsive

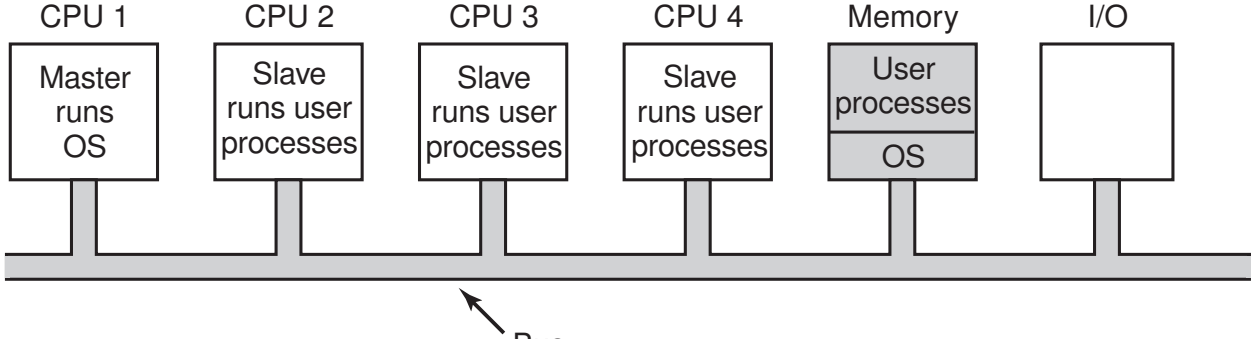
Synchronization on Multiprocessors (1/2)

OS instance per CPU



- When a process makes a system call it is handled by its own CPU

OS on Master CPU also called Master-Slave

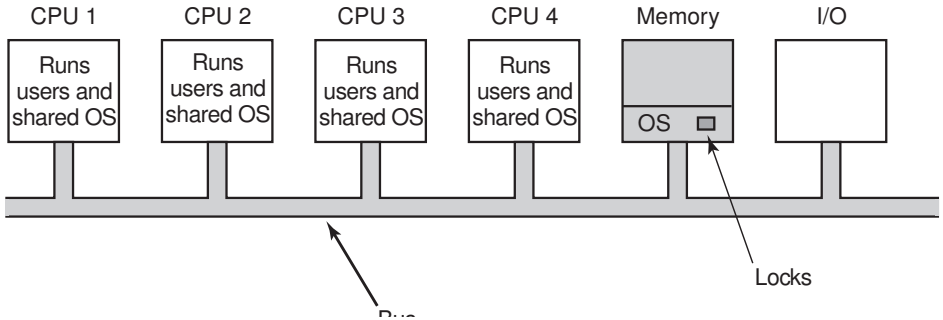


- System calls are redirected to the master CPU

- OS synchronization can be done mostly as on uniprocessors
  - Races can be avoided using nonpremptive kernel

Synchronization on Multiprocessors (2/2)

Single OS which can be run by any CPU



Problem Race conditions

Solution Several:

Single lock whole kernel in the same critical section

- At any time only one process can be inside the kernel
- Requires minimal changes from uniprocessor code

Multiple locks OS components are independent

- However, there are some data structures, such as the process table, that are accessed by otherwise independent parts of the kernel
- Access to multiple data structures may lead to **deadlock**

Agenda

Classic Synchronization Problems

Synchronization in the Kernel

Additional Reading

Additional Reading

Sistemas Operativos

- Section 6.3
- Section 5.6

Modern Operating Systems, 2nd. Ed.

- Section 2.4
- Section 8.1

Operating Systems Concepts, 7th. Ed.

- Subsection 6.6