OS 161: Synchronization Primitives

System Programming - Sarah Azimi

CAD & Reliability Group
DAUIN– Politecnico di Torino

Chapter 6: Synchronization Tools

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores

Background

- Process can execute concurrently or in parallel.

- Coordinating access to shared resources

- Problems
  - Race conditions
  - Deadlocks
  - Resource starvation

- Solutions
  - Synchronization: locks, barriers, semaphores, etc.
  - …

Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has a **critical section** segment of code that is shared with at least one other process
  - Process may be changing common variables, updating the table, writing a file, etc
  - When one process in the critical section, no other may be in its critical section
- ***Critical section problem*** is to design a protocol that the process can use to synchronize their activity to cooperatively share data.

Critical Section

- The section of the code implementing this request is the **entry section**.
- The critical section may be followed by an **exit section**.
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, the remaining code is the **remainder section**

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

General structure of process ***P_i***

Solution to Critical-Section Problem

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. **Bounded Waiting** – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Critical-Section Handling in OS

Two approaches, depending on if the kernel is preemptive or non-preemptive

- **Preemptive** – allows a process to be preempted while it is running in kernel mode.
- **Non-preemptive** – does not allow a process running in kernel mode to be preempted, a kernel mode process will run until it exists kernel mode, blocks, or voluntarily yields control of the CPU.
  - A non-preemptive kernel is essentially free of race conditions in kernel mode

Peterson's Solution

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)
- It is restricted to **two processes** solution that alternate **execution** between their **critical sections** and remainder sections.
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- It requires the two processes to share two variables:
  - **`int turn;`**
  - **`boolean flag[2]`**
- The variable ***`turn`*** indicates whose **turn it is to enter the critical section,** If it is 1 then the process is allowed to execute in its critical section.
- The ***`flag`*** array is used to indicate **if a process is ready to enter the critical section**. **`flag[i] = true`** implies that process $\mathbf{P_i}$ is ready!

Algorithm for Process P_i

- To enter the critical section, the process **p_i set the flag[i]** to be **true** and then sets **turn to the value j**.
- Asserting that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Peterson's Solution

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
- To improve performance, processors and/or compilers may reorder read and write operations that have no dependencies.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Peterson’s Solution

- Two threads share the data:

```
boolean flag =false;
int x = 0;
```

- Thread 1 performs

```
while (!flag);
print x
```

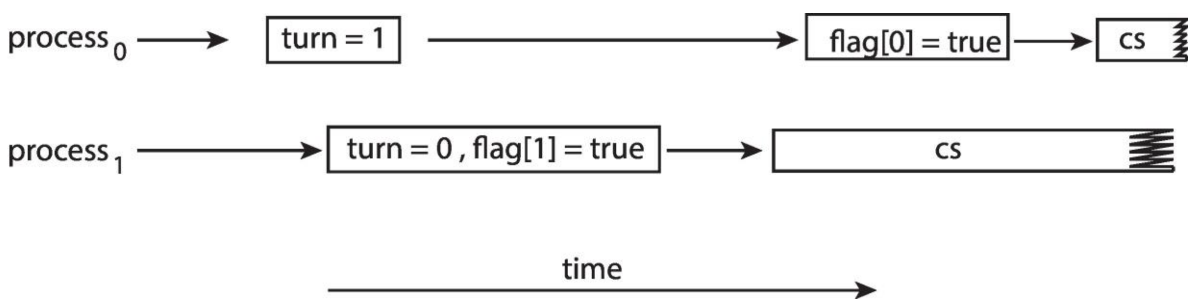
- Thread 2 performs

```
x = 100;
flag = true
```

- What is the expected output? 100

Peterson’s Solution

- 100 is the expected output.
- However, as there are no data dependencies between the variables flag and x, it is possible that a processor may reorder the instruction for thread 2 so that flag is assigned true before assignment of x=100.
  - `flag = true; x = 100;`
- If this occurs, the output may be 0!
- How this affect the Peterson’s solution?



- This allows both processes to be in their critical section at the same time!
- The only way to preserve mutual exclusion is by using proper synchronization tools.

Synchronization Hardware

- Up to now, software-based solution to critical-section problem which do not guarantee to work on modern computer architecture.

- Three hardware instructions that provide support for solving the critical-section problem:

  - Memory barriers

  - Hardware instructions

  - Atomic variables

Synchronization Hardware

- Up to now, software-based solution to critical-section problem which do not guarantee to work on modern computer architecture.

- Three hardware instructions that provide support for solving the critical-section problem:
  - **Memory barriers**
  - Hardware instructions
  - Atomic variables

Memory Barriers

- **Hardware instructions to provide support for solving the critical-section problem:**
- **Memory model:** a computer architecture determines what memory guarantees it will provide to an application program.
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Memory Barriers

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

Synchronization Hardware

- Up to now, software-based solution to critical-section problem which do not guarantee to work on modern computer architecture.

- Three hardware instructions that provide support for solving the critical-section problem:
  - Memory barriers
  - **Hardware instructions**
  - Atomic variables

Hardware Instructions

- Many modern computer systems provide special hardware instructions that allow us to either test and modify the content of a word or to swap the contents of two words **atomically:**

- **Test-and-Set** instruction >> performing a test on a condition, if the condition is true, set a value.
- **Compare-and-Swap** instruction >> comparing the two variables and swapping the values.

test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of the passed parameter
3. Set the new value of the passed parameter to **`true`**

Solution using test_and_set()

testAndSet(lock) algorithm works in this way – it always returns whatever value is sent to it and sets lock to true. The first process will enter the critical section at once as testAndSet(lock) will return false and it'll break out of the while loop. The other processes cannot enter now as lock is set to true and so the while loop continues to be true.

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

           /* critical section */ set lock to true

           lock = false;

           /* remainder section */

 } while (true);
```

compare_and_swap Instruction

Definition:

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

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter **`value`**
3. Set the variable **`value`** the value of the passed parameter **`new_value`** but only if **`*value == expected`** is true. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer `lock` initialized to 0;
- Solution:

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

    /* critical section */ lock = 0;

    /* remainder section */
}
```

Solution using compare_and_swap

Swap algorithm is a lot like the TestAndSet algorithm. Instead of directly setting lock to true in the swap function, key is set to true and then swapped with lock. First process will be executed, and in while(key), since key=true , swap will take place and hence lock=true and key=false. Again next iteration takes place while(key) but key=false , so while loop breaks and first process will enter in critical section. Now another process will try to enter in Critical section, so again key=true and hence while(key) loop will run and swap takes place so, lock=true and key=true (since lock=true in first process).

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

    /* critical section */ lock = 0;

    /* remainder section */
}
```

Synchronization Hardware

- Up to now, software-based solution to critical-section problem which do not guarantee to work on modern computer architecture.

- Three hardware instructions that provide support for solving the critical-section problem:

  - Memory barriers
  - Hardware instructions
  - **Atomic variables**

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) operations on basic data types such as integers and booleans.
- For example, the **`increment()`** operation on the atomic variable
  - **`sequence`** ensures **`sequence`** is incremented without interruption:

Atomic Variables

- The functions are often implemented using compare_and_swap() operation:
- The **`increment()`** function can be implemented as follows:

```
void increment(atomic_int *v)
{
    int temp;

    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v,temp,temp+1));
}
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problems.

- Simplest is **Mutex Lock** for protecting critical sections and preventing race condition:
  - A process must acquire a lock before entering a critical section using **`acquire()`** function
  - Releasing the lock when it exits the critical section using the **`release()`** function
  - A mutex lock has a Boolean variable available whose value indicates if the lock is available or not.
  - If the lock is available, a call to **`acquire()`** succeeds and the lock is considered unavailable.

Mutex Locks

- Calls to `acquire()` and `release()` must be atomic.
  - Usually implemented via hardware atomic instructions such as compare-and-swap.

  - **Disadvantage**: this solution requires **busy waiting,** while the process is in the critical section, the other processes must continuously loop to **`acquire()`**.
  - Wasting CPU cycle.

- This lock is therefore called a **spinlock.**
  - **Advantage:** no context switch is required when a process might wait on a lock. If a lock is to be held for a short duration, one thread can “spin” on one processing core while another thread performs its critical section on one another.

Solution to Critical-section Problem Using Locks

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

OS/161 Spinlocks (kern/thread/spinlock.c)

```
spinlock_acquire(struct spinlock *splk) {
   ...
   while (1) {
      /*  Do test-test-and-set, that is, read first before doing test-and-set, to
          reduce bus contention.
          Test-and-set is a machine-level atomic operation
       */
      if (spinlock_data_get(&splk->splk_lock) != 0) {
         continue;
      }
      if (spinlock_data_testandset(&splk->splk_lock) != 0) {
         continue;
      }
      break;
   }
   ...
}
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- A semaphore **`S`** is an integer variable that, apart from initialization is accessed through two standard **atomic** operations:
  - **`wait()`** and **`signal()`**

- Definition of the **`wait()`**

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

- Definition of the **`signal()`**

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

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Behave similarly to **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**`synch`**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch); S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

OS/161: Disabling Interrupts

- On a uniprocessor, only one thread at a time is actually running.
- If the running thread is executing a critical section, **mutual exclusion may be violated if:**
  1. The running thread is **preempted** (or voluntarily yields) while it is in the critical section
  2. The scheduler chooses a different thread to run, and this new thread enters the same critical section that the preempted thread was in
- Since **preemption** is caused by **timer interrupts**, mutual exclusion can be enforced by **disabling timer interrupts** before a thread enters the critical section, and re-enabling them when the thread leaves the critical section. This is the way that the **OS/161** kernel enforces mutual exclusion.

OS/161: Disabling Interrupts

- There is a simple interface **(splhigh(), spl0(), splx())** for disabling and enabling interrupts.
  - See kern/arch/mips/include/spl.h.

- In reality the **change the priority of the threads**. So, the threads with lower priority cannot interrupt the execution of a thread with higher priority.

OS/161 Semaphores (1.9x: interrupt based)

```
struct semaphore {
   char *name;
   int count;
};

struct semaphore *sem_create(const char *name, int
     initial count);
void P(struct semaphore *);
void V(struct semaphore *);
void sem_destroy(struct semaphore *);
```

Structure of data defining semaphore: name of the semaphore and count variable

Four functions for creating, signaling, waiting and destroying semaphore.

see

- kern/include/synch.h
- kern/thread/synch.c

OS/161 Semaphores: P() (1.9x: interrupt based)

```
void P(struct semaphore *sem) {
   int spl;
   assert(sem != NULL);

   /* May not block in an interrupt handler.
    * For robustness, always check, even if we can actually
   *complete the P without   blocking. */
   assert(in_interrupt==0);

   spl = splhigh();
   while (sem->count==0) {
     thread_sleep(sem);
   }
   assert(sem->count>0);
   sem->count--;
   splx(spl);
}
```

Saving the current priority of the process to recover later and Increasing the priority of the process.
Considering the semaphore value, if the value of semaphore is 0, go to sleep.
If sem is bigger than 0, increasing sem.
Changing the priority to the original one.

OS/161 Semaphores: V() (1.9x: interrupt based)

```
void V(struct semaphore *sem)
{
   int spl;
   assert(sem != NULL);
   spl = splhigh();
   sem->count++;
   assert(sem->count>0);
   thread_wakeup(sem);
   splx(spl);
}
```

Saving the current priority of the process to recover later and Increasing the priority of the process.

Increasing the sem

Waking up one of the processors in the waiting list.

Thread Blocking in OS/161

Implementation of semaphores are based on these two functions.

- OS/161 thread library functions:
  - `void thread_sleep(const void *addr)` blocks the calling thread on address addr
  - `void thread_wakeup(const void *addr)` unblock threads that are sleeping on address addr

- **`thread_sleep()`** is much like `thread_yield()`. The calling thread voluntarily **gives up the CPU**, the scheduler chooses a new thread to run, and dispatches the new thread. However
  - after a `thread_yield()`, the calling thread is *ready* to run again as soon as it is chosen by the scheduler.
  - after a **`thread_sleep()`**, the **calling thread is blocked**, and should not be scheduled to run again until after it has been explicitly unblocked by a call to **`thread_wakeup()`**.

OS/161 Locks

- OS/161 also uses a synchronization primitive called a ***lock***. Locks are intended to be used to enforce mutual exclusion.

```
 struct lock *mylock = lock create("LockName");

lock_aquire(mylock);
  critical section /* e.g., call to list remove head */
lock_release(mylock);
```

- A ***lock is similar to a binary semaphore*** with an initial value of 1. However, locks also enforce an additional constraint: the thread that releases a lock must be the same thread that most recently acquired it.
- The system enforces this additional constraint to help ensure that locks are used as intended.

Condition Variables

- While mutexes implement synchronization by controlling thread access to data, **condition variables allow threads to synchronize based upon the actual value of data**.

- In a critical section, a thread can suspend itself on a condition variable if the state of the computation is not right for it to proceed.

  - It will suspend by waiting on a condition variable
  - It will release the critical section lock (MUTEX)
  - When that condition variable is signaled, it will become ready again. It will attempt to reacquire that critical section lock and only then it will be able to proceed

Using Condition variables with cv_signal

- **Always used together with locks**
  - The lock protects the shared data that is modified and tested when deciding whether to wait or signal/broadcast
- General Usage:

P_i

```
lock_acquire(lock);
while (condition not true)
{
  cv_wait(cond,lock);
}
... // do stuff
lock_release(lock);
```

P_j

```
lock_acquire(lock);
... // modify condition
cv_signal(cond);
lock_release(lock);
```

Using Condition variables with cv_broadcast

P_i

```
lock_acquire(lock);
while (condition_i not true){
  cv_wait(cond, lock);
}
... // do stuff
lock_release(lock);
```

P_j

```
lock_acquire(lock);
...
// modify conditions
// either for Pi or Pk
cv_broadcast(cond);
lock_release(lock);
```

Sending the signal in broadcast at processes that are waiting, waking up all, the one that take the control of CPU first, goes on with the implementation. (to be implemented)

P_k

```
lock_acquire(lock);
while (condition_k not true){
  cv_wait(cond, lock);
}
... // do stuff
lock_release(lock);
```

OS/161 Wait Channels

- Same as condition variables not using semaphores but using spinlocks (busy waiting).
    - Spinlock owned for short time
    - Nested or multiple spinlocks not allowed

- Kernel level synchronization objects

- Integrated with thread scheduling
    - Spinlock handled within thread_switch

OS/161 TODO

- Kernel level synchronization objects to be implemented:
  - Locks
  - Condition Variables

- Strategy:
  - look at semaphores
  - Use spinlocks and wait channels

LAB 3!