

Review: how to share memory

Lecture 8

Process and thread synchronization

In this week and the following weeks, we will see how processes can make use of shared memory frames to communicate with each other.

References:

- [OSC] Section 4.4 (Cooperating Processes), 7.1–7.7 (Process Synchronization).
- [ULK] pp. 24–28 (Synchronization and Critical Regions), 299–310, 314–320 (Kernel Synchronization).

POS(0230A)

POS(0230A)-8.1

The need for synchronization

- **Consider the application of tutorial 2:** Two processes try different passwords to find the one used to encrypt a mail.
- Suppose we use the following scheme: two processes P0 and P1 search from two ends of the search space, i.e., one searching from three spaces towards three tildes, and one the opposite way.
- Note that **both processes** will eventually find the password if not interrupted. Whichever process getting the correct password should thus **notify** the other, say by setting a flag in some shared memory.
- There are three actions to do: **check** the flag, **set** the flag, and **print** out the result. But in what ordering?

POS(0230A)-8.2

The problem: Race conditions

If we **check a common flag before setting**, it is possible that:

- P0 and P1 both find the result.
- P0 and P1 both checks whether the flag is set. Both found it is not.
- P0 and P1 both set the flags and output. Oops.

If we **set a per-process flag before checking**, it is possible that:

- P0 and P1 both find the result.
- P0 and P1 both set their own flag.
- P0 and P1 both think the other prints, and thus don't print. Whoops.

We call such situations **race conditions**: the problem occurs only when certain **timing relations** of different processes hold.

We need some coordination, or **synchronization**, between processes.

POS(0230A)-8.3

Solving the problem with Critical sections

How we can fix the problem?

- We want “setting the flag” and “checking the flag” to be **atomic**.
Of course, “atomic” means it cannot be divided to something smaller.
- E.g., once a process starts to *check* the flag, it guarantees to also have *set* the flag before the other process is able to start checking the flag.
- We say the *set-and-check* code to be in a **critical section**: we cannot allow multiple processes to run that code simultaneously.

Question: *Is there just a few cases when we need critical sections?*

Answer: *No, critical sections are usually needed.* E.g., if two processes need to modify a linked list, we want a big critical section: it is chaotic even if one *read* the list when the other modifies.

Most kernels need a lot of critical sections (why?). In other words, nearly all kernel programs we wrote have synchronization problems.

POS(0230A)-8.4

Definition: critical section

Critical section appears like this:

```
do {
    enter critical section
    // critical section
    exit critical section
    // remainder section
} while (1);
```

Requirements: a solution should have the following properties.

1. **Mutual exclusion:** no two processes are in C.S. simultaneously.
2. **Progress:** if no process is in C.S. and some is waiting, one process is selected to enter, without involving processes not waiting to enter C.S.
3. **Bounded waiting:** there is an upper bound on the number of other processes entering C.S. when one process waits to enter.

POS(0230A)-8.5

A closer look...

- The problem occurs if we have **multiple processors** running the two processes (which is the intention of that tutorial).
- But what if we only have a single processor? Are we immune?
- Not quite... such problems can still arise for **two reasons**:
 - Recall that the OS **switches the processor among processes** to create the illusion of concurrency. It is possible for P0 to be stopped right after it checks the flag but before it sets it. If P1 then executed that code, still have trouble.
 - If one of the two is in **interrupt or signal handler**... it might happen that an interrupt or a signal occurs—just between the test and set.

So we have 3 primary reasons for such problem: multiple processor, involuntary loss of CPU (called “preemption”), and interrupt.

POS(0230A)-8.6

Solving the problem by not using preemption

- There is a simple solution: don't allow preemption.
- This works fine if we have only 1 processor, and the data is **not** shared with interrupt handler.
- But there are two drawbacks:
 - Limited to **kernel mode**. If user mode processes cannot preempt each other, a buggy program is going to monopolize the CPU.
I.e., when you go out of kernel mode, you must allow preemption. It is fine to disallow preemption in kernel mode because kernel functions trust each other.
 - **Slower response time**, since at some time the CPU cannot switch among processes to create illusion of concurrency.
So all critical sections should be made short.

POS(0230A)-8.7

Solving the problem with Interrupt disabling

- There is an even simpler solution: just disallow interrupts!
- This works fine if we have only 1 processor, since **preemption** and **interrupts** both won't happen if we disable interrupts.
- But there are two drawbacks:
 - Limited to **kernel mode**. User processes cannot disable interrupts. (They can disable signals, but that doesn't deal with preemption.)
 - **Costly** and cannot be used too much: if an interrupt occurs and you don't handle it quickly enough, the data indicated by the interrupt could be gone.
The point of interrupts is usually to say “I've some data waiting, please get it as quickly as you can”.
- The approach can't deal with problems that come up from multiple processors. We need another solution.

POS(0230A)-8.8

Software implementation for 2 processes

Software solutions aren't really simple. Let's see a simpler one, that only handle the case for 2 processes P0 and P1.

This solution involves 3 shared flags: *flag*[0], *flag*[1] and *turn*.

```
/* enter critical section of process i */
flag[ i ] = true;
turn = j;
while (flag[ j ] && turn == j);

/* exit critical section of process i */
flag[ i ] = false;
```

Here, P0 has *i* = 0 and *j* = 1, and P1 has *i* = 1 and *j* = 0. *flag*[*i*] is 1 whenever *P_i* want to enter or has entered CS.

Question: why it is a solution? How it guarantees the three requirements of critical sections we just mentioned?

POS(0230A)-8.9

Why it is a solution

The *turn* variable mysteriously makes sure that only one of the 2 processes can enter the CS.

- **Mutual exclusion:** Suppose both processes are in CS. Let P0 finish the while loop first. When P1 executes the while loop, *flag*[0] must be set (otherwise P1 enters CS first), so *turn* must be 1. But *turn* is assigned 0 by P1 before, so P0 must have assigned it to 1 again. But then, P0 can't enter CS. Contradiction.
- **Progress:** If only 1 wants to enter, the other's flag is false, so CS entry is immediate. If both want to enter, *turn* is either 0 or 1, thus one process enters CS.
- **Bounded waiting:** Suppose P0 is waiting when P1 is in CS. Then P0 had finished executing *turn*=1, so P1 can't enter CS again before P0.

But extending it to more processes is not trivial at all. Read the textbook about the *bakery algorithm* (OSC Section 7.2.2).

POS(0230A)-8.10

Hardware solution

- To ease synchronization, multi-processor architectures have instructions that do multiple things **atomically**. E.g.:
 - **Atomic test and set:** check whether a memory location is zero, set the memory location to one, and get the previous value in register, all in a single instruction.
 - **Atomic swap:** swap the content of a memory location with the value of a register in a single instruction.
How? By locking the memory bus, so that other processors can't use memory.

With atomic test-and-set, the following provides C.S. using a shared flag:

```
/* enter critical section of any process */
while (TestAndSet(flag));

/* exit critical section of any process */
flag = false;
```

POS(0230A)-8.11

Quick and dirty way to implement critical section

The solution loops, or “spins”, around the while statement until nobody is in the critical section. We thus call them **spin locks**.

- **Problem 1:** The algorithm does not guarantee bounded waiting. (Why?)
Exercise: Read the textbook to see how to guarantee bounded waiting.
- **Problem 2:** The algorithm wastes CPU during wait.

Such *spin locks* are generally used when **locks are expected to be extremely short**, e.g., shorter than context switch penalty (Again, why?).

Alternative: waiting locks.

- Put the process to **sleep** (waiting state) if the lock is held by others.
- When unlock, check whether some process is sleeping for the lock, and if so **wake** it up.

POS(0230A)-8.12

Example: our simple kernel counter

- Still remember our kernel counter we implement in tutorial 3? In a multi-processor computer, it is possible that many processes try to increment the counter at the same time, resulting in a wrong result.
- Protect it with a spin lock...

```
static int mycounter = 0;
static spinlock_t mycounter_lock = SPIN_LOCK_UNLOCKED;
asm linkage long sys_inccount(void) {
    int newcounter, ret;
    spin_lock(mycounter_lock); // wait until nobody uses mycounter
    newcounter = mycounter + 1;
    if (newcounter < 0)
        ret = -EINVAL; // don't return right away! must unlock
    else
        ret = mycounter = newcounter;
    spin_unlock(mycounter_lock); // unlock it so others can play with the lock
    return ret;
}
// Similar for deccount() and getcount()
```

POS(0230A)-8.13

Generalization: semaphore

To solve more complicated problems, one usually use a more general construct called **semaphore**.

- Each semaphore has a non-negative **count**, with an initial count n , defined when the semaphore is created.
- There are **two operations** on a semaphore: *up()* and *down()*.
- *down()* **waits** until count is non-zero. Then it **decrements** the count.
- *up()* does what the name suggest: **increment** the count.

Note that a critical section is created using a semaphore with $n = 1$. We usually call such semaphore a **mutex (mutual exclusion)**.

Or “binary semaphores”, since the value is either 0 or 1.

Semaphore can also be used to **wait for some other process**: Set $n = 0$ initially. A process calling *down()* will block until a process calls *up()*.

POS(0230A)-8.14

User level synchronization: semaphore and locking

Waiting locks involve putting processes into sleep. For user processes, they must be supported by the kernel. This is achieved in two ways:

- **User-level semaphore.** E.g., SysV or POSIX semaphore construct. They exports the semaphore construct to user land.
Read the tutorial notes!
- **File and record locking.** We can use *flock()* or *lockf()* to lock a file for reading or writing. This is like a *mutex*, providing partial solution for synchronization. Again, we see the filesystem is used for addressing.

POS(0230A)-8.15

Performance and practice

3 problems: preemption, interrupt, multiprocessor. Solutions:

Solution	Problem	Local?	Monopolize CPU?	Speed?
No Preemption	Preemption	No	Yes	Fast
Disable Interrupts	Interrupts/Preemption	Yes	Yes	Fast
Spin locks	Multiprocessor	Yes	Mostly	Fast
Waiting locks	Multiprocessor /Preemption	Yes	No	Slow

Local means that we have to identify the critical sections in the program, making it more tedious to code. This is not needed in “no-preemption”.

All kernels disable interrupts to deal with interrupt problem. Some use locks to deal with the remaining problems, while some (uniprocessor) kernels simply disable preemption.

POS(0230A)-8.16

Language constructs: monitors

All locks requires explicitly placing lock and unlock instructions, which is quite error-prone. Can the **compiler** help us to lock automatically?

- Shared data is manipulated using special objects, called **monitors**. Like objects, they are accessed only through **methods**.
- The language guarantees that at any time, **at most one process actively executes** a method of each monitor. If another process invokes a method, it waits. This provides **implicit synchronization**.
- A method can **wait for a condition**. This makes the execution *inactive*, allowing other processes to invoke a method. The programmer of the method must keep the shared data in a good state before waiting.
- A monitor method can **signal a condition**, thus allowing at most one process waiting for the condition to continue execution, after the currently executing method returns or wait.

POS(0230A)-8.17

Example use of monitor

```

monitor counter {
    int inc_count() {
        // won't be called twice
        // at the same time
        int new_count = count + 1;
        if (new_count <= 0)
            return -1; // error
        count = new_count;
        if (count == 2)
            count2_cond.signal();
        return count;
    }

    void wait_count() {
        // exercise: why need to be "while"?
        while (count != 2)
            // wait, must give up lock now
            count2_cond.wait();
        cout << "count is 2 now\n";
    }
}
// data members
int count; // initialized to 0 by constructor
condition count2_cond;
};

```

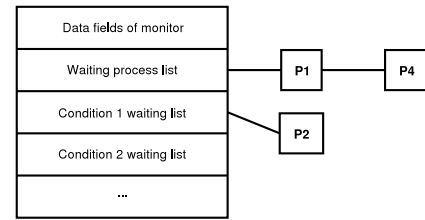
The language guarantees that methods won't run concurrently.

Note that the language doesn't specify what is the meaning of the conditions. The meaning is implied by the way we signal them.

E.g., here we raise it if the count becomes 2. So the meaning of *count2_cond* is that "the value of count becomes 2".

Pictorial view

Conceptually, a monitor is implemented like this:



A process waiting in the waiting process list is ready to execute, once the method currently running in the monitor returns.

A process waiting in the condition waiting list is **not** ready to execute, and must wait until some other process *signals* the condition.