Distributed systems

Author: Daria Shutina

23-09-12

This course is about distribute computing: independent processes that communicate.

One of the key characteristics is concurrency. Single-processor performance does not improve. But we can add more cores and run code concurrently.

Amdahl's Law

The **speedup** is the ratio between the sequential time and the parallel time taken for executing a task.

p - fraction of the work that can be done in parallel

n - number of processors

S - the maximum speedup - measures how much faster the program becomes when parallel processing is applied.

$$S = \frac{1}{1 - p + \frac{p}{n}} < \frac{1}{1 - p}$$

For example, if 90% of a program can be parallelized ($p = 0.9$), and you have 10 processors ($n = 10$), Amdahl's Law would indicate that the maximum speedup achievable is approximately 5.26 times faster than the sequential execution:

$$S = \frac{1}{0.1 + \frac{0.9}{10}} = 5.26$$

Mutual exclusion

A **critical section** (CS) is code that must be executed by only one thread or process at a time. **Mutual exclusion** is a technique that ensures that only one process gets access to CS (mutexes are used to implement it).

- deadlock-freedom: **at least one** process eventually enters its CS. It means that the system prevents deadlocks from occurring. One common approach to achieve deadlock-freedom is to require that threads always acquire mutexes in a specific order.
- starvation-freedom: **every** process eventually enters its CS. It means that the system ensures that no thread or process is continuously blocked from making progress.

Remote memory references (RMRs)

Remote memory references can access data that is stored in a location physically distant from the device which needs to manipulate that data.

Data can be distributed across multiple servers or nodes for scalability or fault tolerance. RMRs are crucial for enabling communication and data sharing among components that are physically separated.

How to measure complexity of the distributed algorithm, where processes may need to busy-wait? *Amount of RMRs is a more realistic measure*.

Peterson's lock

The algorithm is used to coordinate access of two processes to a shared resource. It relies on two shared variables called `turn` and `flag`.

Peterson's lock: 2 processes

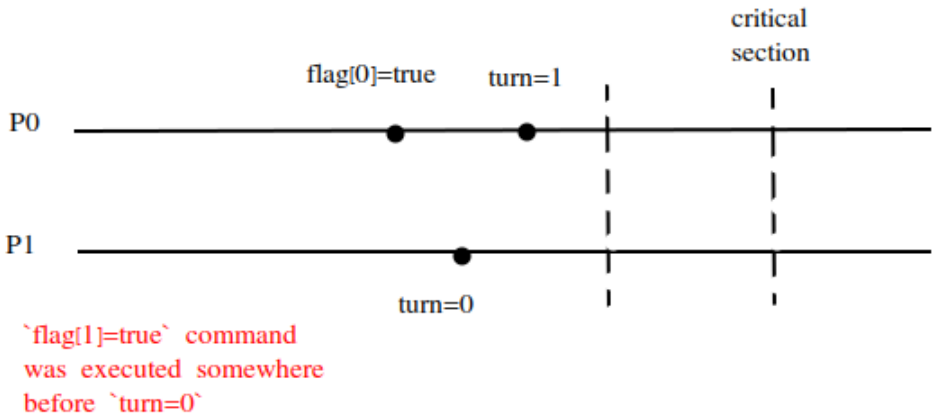
```
bool flag[0]   = false;
bool flag[1]   = false;
int turn;
```

```
P0:

flag[0] = true;
turn = 1;
while (flag[1] and turn==1)
{
          // busy wait
}
// critical section
…
// end of critical section
flag[0] = false;
```

```
P1:

flag[1] = true;
turn = 0;
while (flag[0] and turn==0)
{
          // busy wait
}
// critical section
…
// end of critical section
flag[1] = false;
```

Proof for safety:

We know that the last write operation (before entering the critical section) was changing the `turn` variable. Assume it was `P0` process who assigned `1` to `turn`:

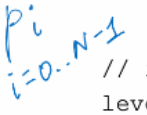


Then there are only read operations before entering the critical section, which means values of `turn` and `flag` variables do not change. `flag[0] and turn==0` condition turns out to be `true`, and P1 enters the critical section. `flag[1] and turn==1` is false, and P0 waits for P1 to finish.

Attempts to change the code:

1. Each process assign another process's ID to `turn` variable. If the process assigns its own ID to `turn`, then both processes will enter the critical section.
2. If we change the sequence of commands `flag[0]=true` and `turn=1`, then both processes will enter the critical section.

Peterson's lock: N ≥ 2 processes




```
// initialization
level[0..N-1] = {-1};      // current level of processes 0...N-1
waiting[0..N-2] = {-1}; // the waiting process in each level
                           // 0...N-2

// code for process i that wishes to enter CS
for (m = 0; m < N-1; ++m) {
    level[i] = m;
    waiting[m] = i;
    while(waiting[m] == i &&(exists k ≠ i: level[k] ≥ m)) {
        // busy wait
    }
}
// critical section
level[i] = -1; // exit section
```

We can understand `level` array as an array of waiting rooms: there can be zero or one process waiting in one room. And `waiting[m] = i` means the process `i` is waiting in the room `m`.

When the process `i` wants to enter the critical section, for every `m`, it first enters the room `m` (`level[i] = m`). Then we mark that there is a process waiting in room `m` (`waiting[m] = i`).

In simple words, the process `i` enters the critical section only when `level[i]` appears to be the maximum value in the `level` array. After `N-1` iterations, only one process satisfies this condition.

Proof for safety:

If processes are at the different stage of iteration over `m`, they cannot enter the critical section simultaneously, since the first condition of the while-loop will be true for the process with the lower `m` value.

Lets consider the situation when there are processes `i_1, ..., i_k` and `level[i_1] = ... = level[i_k] = m` for some `m`. Assigning value to `waiting[m]` was the last write operation, assume we put `i_2` value into it. Then it means that the process `i_2` is stuck in the while-loop, and $k-1$ processes reach the next iteration.

There are $N-1$ iterations in total, so, int the end, only one process reaches the critical point.

Complexity:

The complexity is $O(n^3)$ RMRs per process.

Bakery algorithm

Bakery [Lamport'74,simplified]

```
// initialization
flag: array [1..N] of bool = {false};
label: array [1..N] of integer = {0}; //assume no bound

// code for process i that wishes to enter CS

flag[i] = true; //enter the "doorway"
label[i] = 1 + max(label[1], ..., label[N]); //pick a ticket
//leave the "doorway"
while (for some k ≠ i: flag[k] and (label[k],k)<<(label[i],i));
// wait until all processes "ahead" are served
…
// critical section
…
flag[i] = false; // exit section
```

Processes are served in the "ticket order": first-come-first-serve