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:                              P1:


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

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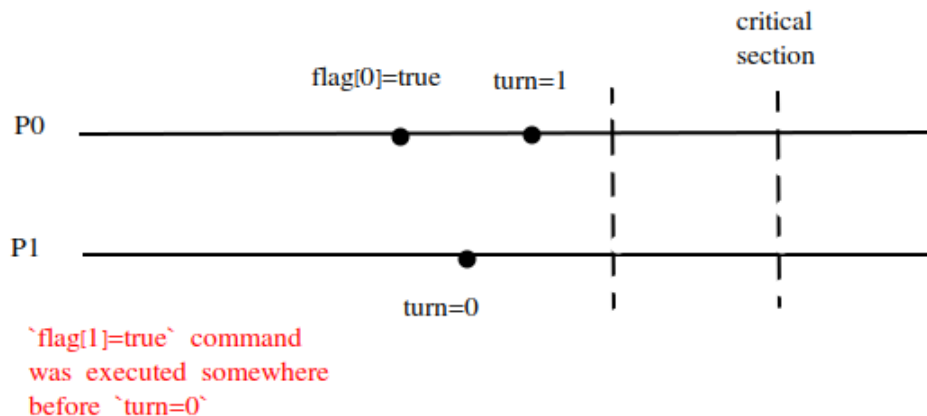
Then there are only read operations before entering the critical section, which means values of `turn` and `flag` variables do not change. `flag[0] and turn==0` condition turns out to be `true`, and P1 enters the critical section. `flag[1] and turn==1` is false, and P0 waits for P1 to finish.

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 $O(n^2)$ RMRs.

n-process tournament tree

The leaves are the processes that what to enter the critical section. The process which gets to the top of the tree receives the right to do it.

The complexity is $O(\log N)$, since, in every node, Peterson's lock for two processes is executed, which means a constant amount of RMRs. The hight of the tree is $\log N$.

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

23-09-19

Abstractions

The **system model** is the environment of our algorithm. The model should be simple and tractable.

Process abstraction – an entity performing independent computation. In this course we deal with deterministic processes (based on the current state and the input, the process takes one particular step).

Processes share common characteristics. Each of them has states, inputs, outputs and a sequential specification [1]. A distributive algorithm is a collection of such processes.

Communication abstraction – includes shared memory and message-passing. Shared-memory objects also have states and interfaces. Interfaces describe operations you can apply on these objects and responses the objects may provide. They also have a sequential specification (current state + operation = new state & corresponding response).

Examples:

- read-write registers (simply a shared variable)

- TAS (test and set): read and write are combined in one atomic operation. It is assumed that the variable is binary. Value of the variable is first checked, then changed depending on the result.

- CAS (compare and swap): the operation is atomic and takes two parameters. If the value of the variable is equal to the first parameter, then it is replaced with a value of the second parameter.

- etc

Implementing an object

Imagine we want to implement a shared queue. The queue is not implemented in the hardware, instead we have base objects (read-write registers, TAS CAS etc).

We have a FIFO order. `NQ` operation puts an element at the end of the queue, and `DQ` operation takes the first element. Each operation is a sequence of commands which involve changing the state and accessing the shared data.

Both operations need time to finish, and they can overlap in time. So we need to define what it means for an operation to be correct.

Correctness

- **Safety** - you classify states reachable by the algorithm into bad and good states, and you require your algorithm never reaches bad states.

  reach a bad state = there is a *finite* sequence of step which ends up in that bad state.

- **Liveness** - your require that eventually something good should happen.

  can be violated only in an *infinite* execution.

Any automaton should satisfy both safety and liveness properties.

Processes access a communication abstraction by invoking operations. An operation is implemented using a sequence of accesses to base objects. A process is **correct** if, taking infinitely many steps, it never fails in the middle of the operation.

Runs and properties

A **system run (trace)** is a sequence of events (actions the process may take). Every event is atomic access to the shared data or invocation or response.

Σ - **event alphabet** (a set of all possible events).

Σ^ω (Σ^∞) - the **set of** finite (infinite) **runs** = set of finite (infinite) sequences of Σ.

A **property** P is a subset of Σ^ω. An implementation satisfies P if every its run $\in P$.

$e_1, \ldots, e_k$ are events. $\sigma = e_1 \ldots e_k$ is a system run.

P is a **safety property** if:

- prefix-closed: $\sigma = e_1 \ldots e_k \in P \Rightarrow \forall l \leqslant k \ \sigma' = e_1 \ldots e_l \in P$.
- limit-closed: for each infinite sequence of traces $\{\sigma_n\}$, such that $\forall i \ : \sigma_i$ is a prefix of $\sigma_{i+1} \wedge \sigma_i \in P$, then $\sigma = \lim\limits_{n \to \infty} \sigma_n \in P$.

P is a **liveness property** if every finite trace σ has an extension in P.

$$\forall \sigma \ : \ \sigma = e_1, \ldots, e_k \in \Sigma^\omega \ \Rightarrow \ \exists \sigma' = \underbrace{e_1, \ldots, e_k, e_{k+1}, \ldots}_{\text{extension of } \sigma} \in P.$$

A SIDE REMARK: Safety makes the process closed, and liveness makes it dense. Every property is an intersection of a safety property and a liveness property:

$$\forall P \subseteq \Sigma^\omega \ \exists S, L \in \Sigma^\omega \ : \ P = S \cap L$$

How to distinguish safety and liveness

P is a property (a set of runs)

- If every run that violates P is *infinite*, then P is liveness.
- If every run that violates P *has a finite prefix* that violates P, then P is safety
- otherwise, P is both

Examples

We have a collection of processes. They have inputs and outputs. Their tasks are to propose values and decide on values (it does not matter what these operations actually do, forget about the sense).

So, we have two kinds of events: $\texttt{propose}_\texttt{i}(v)$ and $\texttt{decide}_\texttt{i}(v)$ - "propose/decide value $\texttt{v}$" and events are performed by the process $\texttt{i}$.

$$\Sigma = \bigcup_{i,v}\{\texttt{propose}_\texttt{i}(v),\ \texttt{decide}_\texttt{i}(v)\}\ \cup \left\{\begin{array}{l}\text{base-object accesses (read,}\\ \text{write) to the shared memory}\end{array}\right\}$$

Property example 1: "Every decided value v was previously proposed" ($\texttt{decide}_\texttt{i}(v)$ happens after $\texttt{propose}_\texttt{j}(v)$ for some processes $\texttt{i}$ and $\texttt{j}$).

It is a safety and liveness property.

Property example 2: No two processes decide differently (means all processes should agree on the same value when making a decision).

It is safety and not liveness.

Property example 3: Every correct (taking infinitely many steps) process eventually decides.

It is not safety, but liveness.

Property example 4: No two correct processes decide differently.

It is safety and liveness.

Linearizability (atomicity)

When an operation is invoked, it should take effect instantaneously. In other words, there should be no perceptible delay between the invocation of the operation and when its effect becomes visible.

Linearizability allows the operation to take some period of time (=not exactly atomic).

Histories

A **history** is a sequence of invocations and responses. E.g. the history is `p1: enq(0)`, `p1: ok`, `p3: deq()`, `p1: enq(1)`, `p3: 0`, `p3: deq()`, `p1: ok`, `p2: deq()`, `p2: 1`.

Example 1
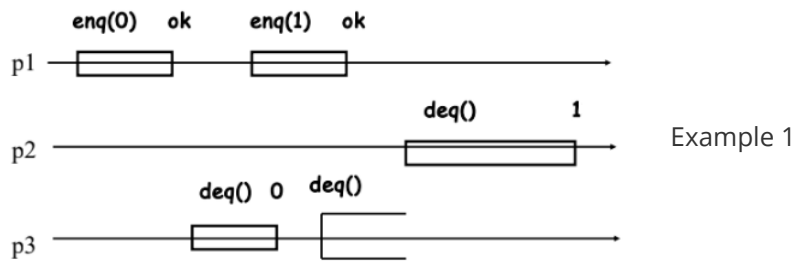
It is **sequential** if every invocation is immediately followed by a corresponding response. A sequential history has no concurrent operations. E.g. a sequential history is `p1: enq(0)`, `p1: ok`, `p3: deq()`, `p3: 0`, `p1: enq(1)`, `p1: ok`, `p2: deq()`, `p2: 1`, `p3: deq()`.

Example 2
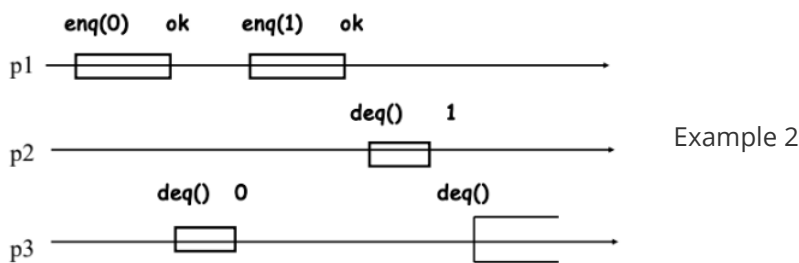
A sequential history is **legal** if it satisfies the sequential specification of the shared object (FIFO policy is used for queues or, for read-write registers, it is true that every read operation returns the last written value).

Let H be a history.

An operation p is **complete in** H if H contains both invocation and response of p.

A completion of H is a history H' that includes all complete operations of H and a subset of incomplete operations of H followed with matching responses.

---

1. A sequential specification is a detailed description of how a process or system operates step by step. For example, it can be FIFO policy or read-write rule: Every read returns the last written value. ↩