CMPSCI 590S Systems for Data Science **Fall 2017**

Lecture 5

Lecturer: Emery Berger *Scribe: Aishwarya Sudhakar, Kevin Feveck*

5.1 Processes

In UNIX land, to create a process we invoke `fork()` and the calling process is cloned. In the parent process, `fork()` returns the child's process ID (PID) while in the child process `fork()` returns 0. The child is a clone of the parent and resumes execution at the point after the call to `fork()`, however the child does not share any state with the parent. Each process has it's own memory space.

For processes that are allocated a lot of memory, cloning can be expensive, especially as this memory copy must be done atomically to ensure that the values in memory do not change before they are copied to the child process state. This can take even longer if some memory in the parent process is paged out to disk. This idea of copying all of memory during the clone phase is called pre-emptive or eager copying.

In actuality, this is not how `fork()` is implemented. It instead does what is called on-demand or lazy copying. Only what is needed is copied. Memory pointers, rather than actual values, are copied. When a parent or a child wants to modify a value in memory, a new page is allocated, the values copied to this new page, and then the modification is allowed to happen. This is called a copy-on-write, Copy-on-write ensures that the view of memory only changes for the process that writes/modifies, the other process will still point to the old value.

5.2 Processes vs Threads

Since process creation implements a copy-on-write protocol, there is only a little more overhead when comparing process creation to thread creation. When should we use each? It depends.

Threads communicate via shared memory. Processes do not share memory, and instead communicate via message passing (this is called Inter-Process Communication).

One downside to threads, is that if one thread goes down, due to a runtime error for example, the entire process running the threads is killed. This is not very good for designing fault-tolerant applications. With processes, if a process dies, it does not affect the other running processes. This 'dead' process, can simply be restarted (by a master process for example).

5.3 Tolerating Errors

Say a node has a failure rate of:

$$P_n(fail) = \frac{1}{100}$$

And node failures are independent of one another. If we add another node for redundancy purposes, the probability of failure of the system becomes:

$$P_s(fail) = P_n(fail) x P_n(fail) = \frac{1}{100} x \frac{1}{100} = \frac{1}{10000}$$

If we were to add a third redundant node (Trimodular redundancy), the failure rate of the system becomes:

$$P_s(fail) = \frac{1}{1000000}$$

In addition to redundancy, check-pointing can be implemented so that failed processes can be restarted from a previously known good "snapshot".

Random errors (due to cosmic rays for example) may corrupt bit values in memory. While there are mechanisms for dealing with this in memory (Parity bit, Error Correcting Code (ECC), Single Error Correct Double Error Detect (SECDED)) we can also take a majority vote among our redundant systems. It is unlikely that two will fail, especially in the same way, unless the error is deterministic.

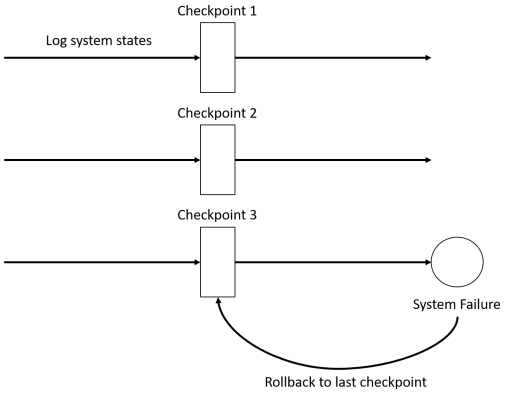
There are two types of bugs that may error in a system: a Bohr bug (deterministic) and a Heisenbug (non-deterministic). Bohr bugs are good from the programmer perspective because they are reproducible and can be easier to debug. This is not the case with Heisenbugs; they are much harder for the programmer to reproduce and debug. From the user perspective, deterministic errors mean the system will always fail under these conditions, and this is a bad thing. Heisenbugs are much more tolerable, since the process/task can be restarted and will probably complete this time.

5.4 How to increase reliability

The notion of reliability with independent systems is achieved through a backup. There are different kinds of backup:

- **Periodical Backup** At defined checkpoints, a snapshot of the system is stored in disk. The snapshot includes the state of the system from the last successful checkpoint. This is also called incremental backup

- **Full Backup** At defined checkpoints, a snapshot of the entire system is taken and stored in disk.

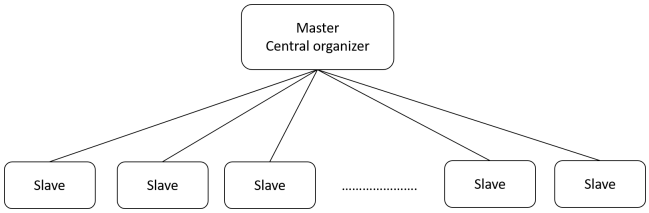


When a failure is incurred, the system rolls back to the last checkpoint, restores its state from the backup and continues processing. There is no significant loss in data. The machines use logs to record the state of the system.

5.5 Master-Slave Architecture

One of the most common architecture of distributed systems is the Master-Slave architecture. This has now been rephrased to Master-Worker architecture. This design includes one of the systems which takes care of the other systems in the cluster. The master is fundamental in scheduling the tasks and also in verifying the health of a worker in the cluster. The master periodically pings the workers of their status, if the worker machines fail to respond, the master then notes these machines as dead and reschedules the tasks assigned to the dead machine to another machine in the cluster.

$$P_m(fail) = \frac{1}{100}$$
$$P_m(sucess) = 1 - \frac{1}{100} = \frac{99}{100}$$



The odds of a master failing are negligible. But in cases where the master fails, the other workers in the cluster hold a **Leader-Election** process, where the workers contend amongst each other to decide who becomes the next master.