```
:(){ :|:& };:
```

Do not actually run this

Process pairs

- Faults may cause Errors
- Unhandled Errors may cause Failures

- What can we do to prevent failures?

- Two approaches to reliability:
  - Remove faults with aggressive testing
    - Impossible for large systems
  - Create programs that can deal with faults
    - What if this isn’t good enough?
    - There will still be errors, and they must not cause failures

- This lecture is not about error detection
  - Let’s assume we already have this
- This is about errors we cannot detect
  - Such as a program crashing for any reason

Extra benefits

- Any fault is less severe than a program crashing
- If we can deal with a crash, we can deal with anything

- Merging of error modes

  An error may cause one type of failure: The program dies

  Possible solution to other errors: Kill the program?

- How do we deal with a crash?

  We need to bring a program back to life

- Two approaches:

  Have a second program monitor the main program,
  and restore it when it dies

  Have the main program create its own backup clone

  This approach is called process pairs

Dealing with the undead

- Programs usually die for a reason

  Restoring the full state of a program that died may cause it to die

    Again.

    And again.

  You will need a well-tested fail-safe state for when a full restore fails

  Pro tip: Have a way of logging the state of a program that crashes

    Go on an exiting treasure hunt. For failures.

Dealing with the undead II

- You may need to kill an otherwise unkillable program

  Read up on using

```
pkill -f progName
kill -9 $(pgrep progName)

taskkill /F /IM progName /T
```

Creating the undead

- The backup must be spawned before the primary dies

  A dead program can’t spawn a program (because it’s dead)

  ⇒ The backup must take an active part

- The backup must not take over before the primary dies

  Otherwise it will create a third program ⇒ Fork bomb

- Perhaps: When a backup takes over, have it kill all other instances of the program (just to be safe)

- General structure:

  The primary broadcasts that it is alive (say, 10 times a second)

  When the backup has lost `n` messages, it takes over

  The backup becomes the primary, and spawns its own new backup

  The cycle continues

Sharing I

- If you need to restore state, this state must be shared

- Checkpoint-restart: The state is placed in common storage. The backup only reads when it takes over. IAmAlive-messages contain nothing.

  Common storage: Other elevators?

Sharing II

- Checkpoint message: The state is sent as the contents of the IAmAlive-message. Current state is the most recent message.

Keep old messages, in case the most recent one caused the failure?

Sharing III

- Persistent: The backup starts in a null state. The transaction mechanism deals with the clean-up

  Requires that the transaction mechanism exists

Watchdogs

- With multithreading, a program may be half-dead
  - One thread may be unresponsive
  - Several may be deadlocked

  These are failures that aren’t merged into crashing the program

- The way to detect this:
  More IAmAlive-messages! (for MP systems, at least)
- Where should these messages go?
  “Thread pairs” are difficult to implement
- We have process pairs...
  Let’s merge

- All threads send IAmAlive-messages to a watchdog
- If the watchdog misses `n` messages, kill the entire program

- What if the watchdog dies?

  Have the watchdog send the IAmAlive-messages to the backup process pair

Everything is a server

- Always use for-select / while(true) receive();
- After every received message, send IAmAlive to watchdog
- Set a timeout on selecting/receiving

  Send a message even if nothing happens

- Note:

  This imposes a real-time demand on handling a message

Logging with watchdogs

Instead of sending IAmAlive after having handled a message, send the content of the message before handling it

(Send the string representations of the types received)

Have the watchdog log the sequence of messages, and go on another treasure hunt for faults

Replay the sequence of messages with MP dependency injection