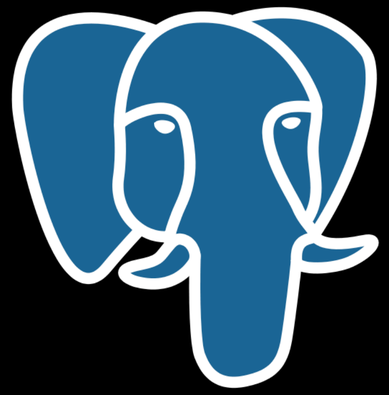


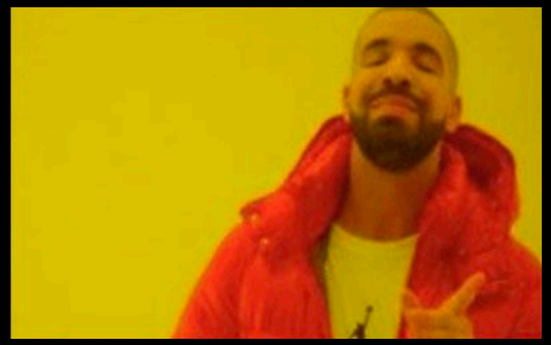
# IPC in PostgreSQL

## Doing things at just the right time

**Thomas Munro | Open source database hacker at Microsoft | PGCon 2023**

# Two decades of improvements

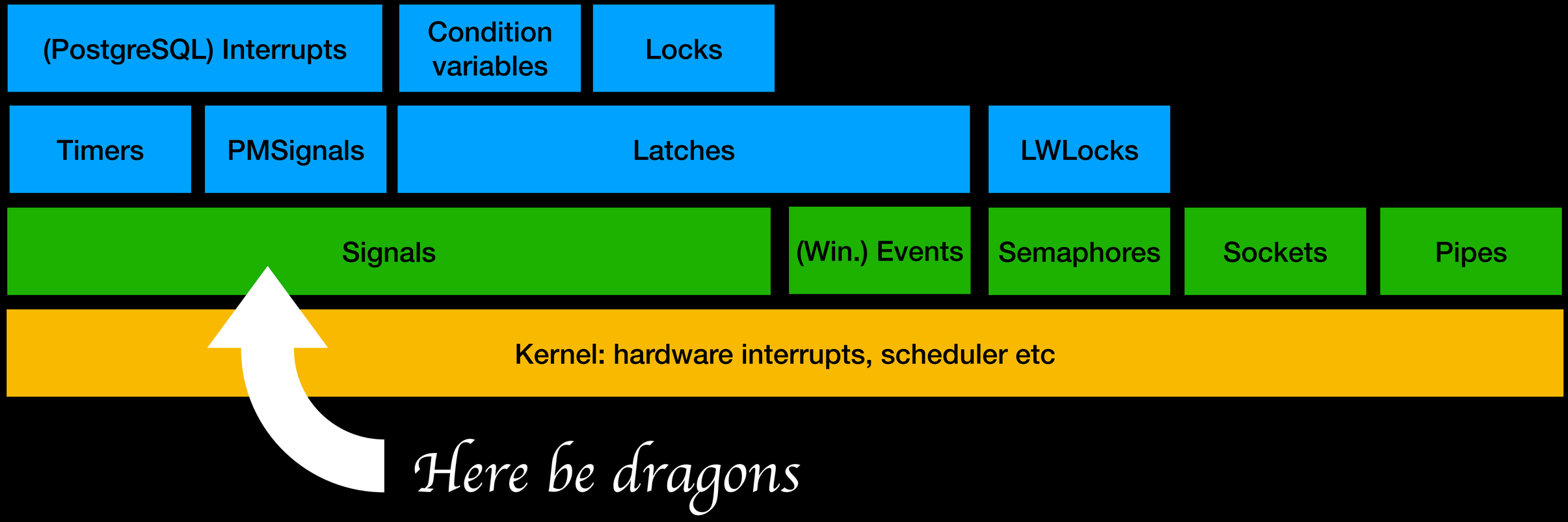
- Sleep/poll loops
- Blocking system calls, expecting signals to interrupt them, unreliably
- Sleeping/waiting without checking for postmaster exit
- Signal handlers doing quite a lot of work
- CHECK_FOR_INTERRUPTS() for “cancel” and “die”

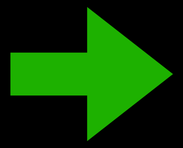


- Non-blocking sockets
- WaitLatch() or WaitEventSetWait() as primary waiting mechanism
- Carefully computed timeouts
- Signal handlers just setting flags and latches
- CHECK_FOR_INTERRUPTS() doing various other co-operative tasks
- More work needed!

# PostgreSQL 16

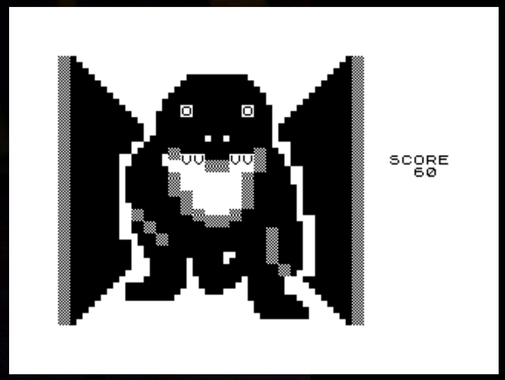
- Postmaster no longer runs state machine and forks children inside a signal handler; this was questionable (and incidentally broke on two obscure OSes)
- [Pending] Recovery conflicts should not be handled in the SIGUSR1 handler!
- Walreceiver no longer wakes up 10 times per second to check for work to do
- Startup process no longer wakes up every 5 seconds to check for promote_trigger_file
- CHECK_FOR_INTERRUPTS() added to various slow code paths

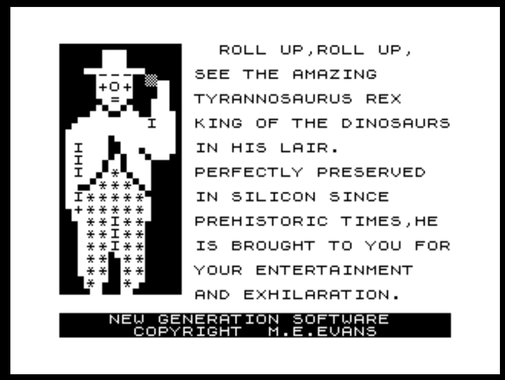

(PostgreSQL) Interrupts
Condition variables
Locks
Timers
PMSignals
Latches
LWLocks
Signals
(Win.) Events
Semaphores
Sockets
Pipes
Kernel: hardware interrupts, scheduler etc
Here be dragons



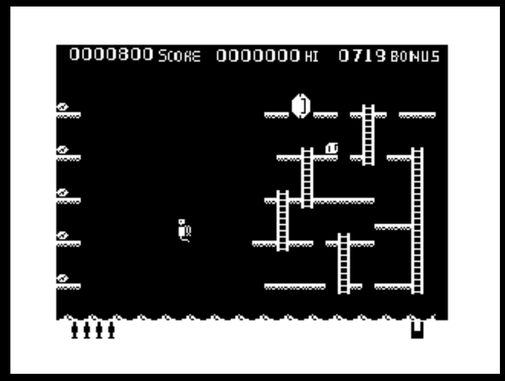
Part I: Signal handlers are dangerous
Part II: Modern PostgreSQL IPC APIs
Part III: Some ideas for future improvements

# Hardware interrupts (very briefly)





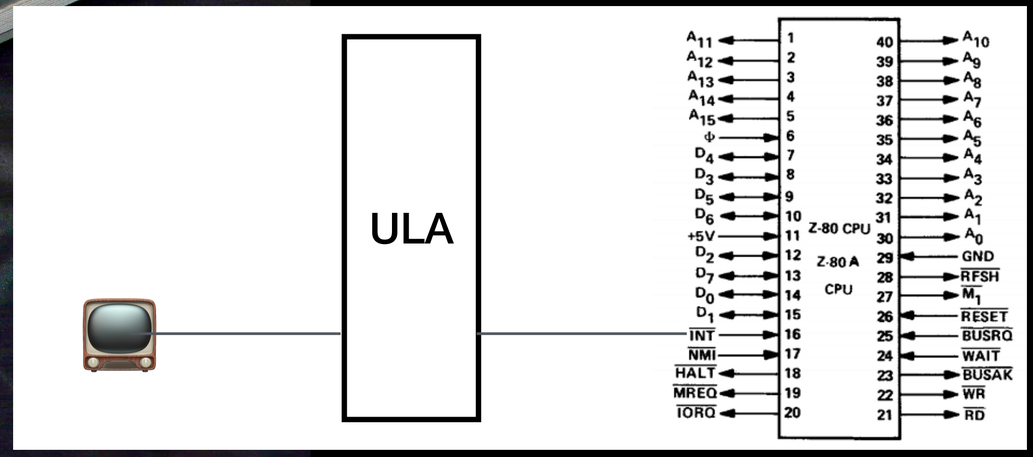
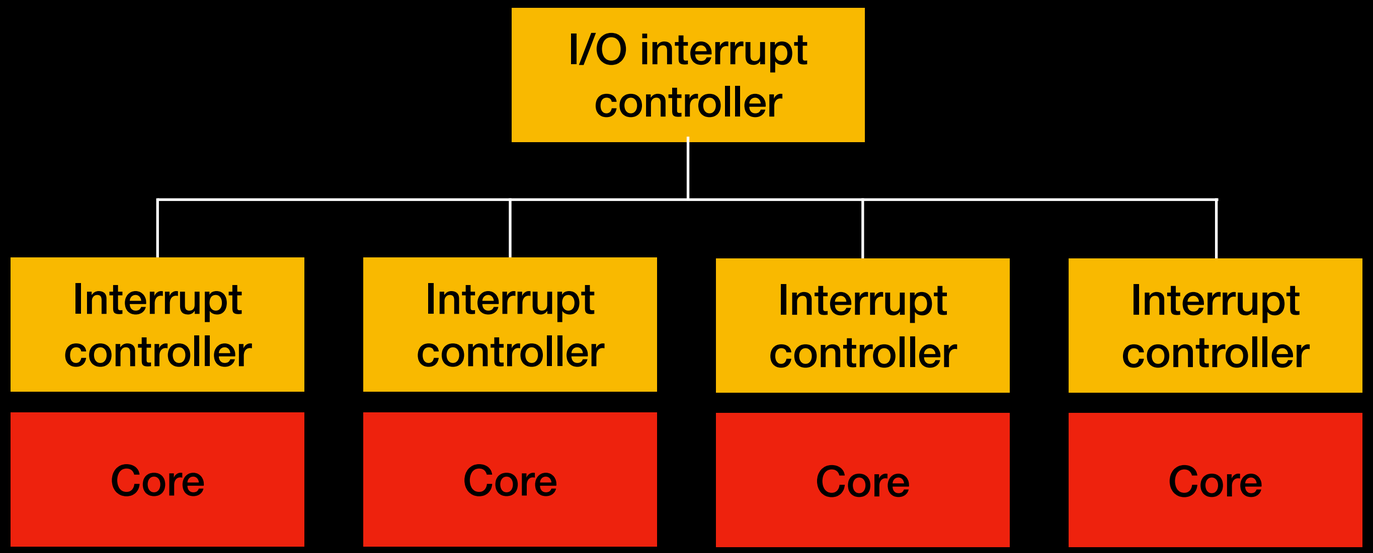




Photo: Mike Cattell, CC-by-2.0, from Wikipedia/ZX81




- I/O
  - Storage
  - Network
  - …
- Timers
  - Time sharing
  - Syscall timeout
  - Interval timer
- Inter-processor (IPI)
  - Wakeup
  - User signals

Interrupts push a thread directly into the kernel*, which creates the illusion of synchronous I/O and multi-tasking, so mostly we don’t care up here in user space, BUT:

*But see recent Intel invention SENDUIPI, user space IPI (not yet exposed by any OS?)

Signals are a technique used to notify a process that some condition has occurred. A signal is similar to an interrupt in that it can cause a process to be involuntarily interrupted. The difference between an interrupt and a signal is that an interrupt is caused by some event external to the processor (a disk I/O completes, a character arrives at a terminal, etc.), whereas a signal is caused by some event internal to the processor (a timer expires, an illegal instruction is executed, etc.). **We can think of signals as software interrupts.**

| | Name | Default Action | Description |
|---|---|---|---|
| 1 | SIGHUP | terminate process | terminal line hangup |
| 2 | SIGINT | terminate process | interrupt program |
| 3 | SIGQUIT | create core image | quit program |
| 4 | SIGILL | create core image | illegal instruction |
| 5 | SIGTRAP | create core image | trace trap |
| 6 | SIGABRT | create core image | abort program (formerly SIGIOT) |
| 7 | SIGEMT | create core image | emulate instruction executed |
| 8 | SIGFPE | create core image | floating-point exception |
| 9 | SIGKILL | terminate process | kill program |
| 10 | SIGBUS | create core image | bus error |
| 11 | SIGSEGV | create core image | segmentation violation |
| 12 | SIGSYS | create core image | non-existent system call invoked |
| 13 | SIGPIPE | terminate process | write on a pipe with no reader |
| 14 | SIGALRM | terminate process | real-time timer expired |
| 15 | SIGTERM | terminate process | software termination signal |
| 16 | SIGURG | discard signal | urgent condition present on socket |
| 17 | SIGSTOP | stop process | stop (cannot be caught or ignored) |
| 18 | SIGTSTP | stop process | stop signal generated from keyboard |
| 19 | SIGCONT | discard signal | continue after stop |
| 20 | SIGCHLD | discard signal | child status has changed |

… usually 32 traditional/reliable signals, with OS variations, and then maybe ‘real time’ signals, not discussed in this talk.

# Who sends signals, and why?

- Standard Unix “outside world” signals: SIGINT for ^C, SIGHUP for reload, SIGTERM for shutdown
- Interprocess requests
  - Shutdown, reload
  - Ad-hoc use of signals with special meanings to certain backends: SIGUSR1, SIGUSR2
  - “PMsignals”: flag + SIGUSR1 to postmaster
  - “Procsignals”: flag + SIGUSR1 to backend
  - Latches: backend SIGURG
- Kernel problems: OOM KILL, FPE, ILL, BUS…
- Kernel: timer -> SIGALRM, child exit -> SIGCHLD, parent exit

# When do signal handlers run?

- *Synchronous* signals are caught immediately because of something the code did (42/0 → SIGFPE, writing to a closed pipe → SIGPIPE, __crc32cb() → SIGILL, …)
- When an *asynchronous* signal is generated it is marked as pending in a process/thread; imagine a bitmap of pending signals
- If it was blocked with eg sigprocmask(), it runs when next unblocked
- Older Unix system would check for pending signals only while rescheduling and at sys call entry/exit (including EINTR, interrupting sleeping system calls)
- Modern SMP Unix systems also use an IPI to interrupt an already-running thread, so could be between any two machine code instructions
- On Windows, we emulate signals in the backend, checking for queued up pseudo-signals to deal with at key places (pgwin32_dispatch_queued_signals())

# Gallery of signal hazards

# Language problems

## Atomicity of loads and stores

```
very_wide_type_t x;

void
signal_handler(int signo)
{
    x = 0;
}

void
f(void)
{
    if (x == 0)
        x = -1;
}
```

(Made-up pseudo-assembler!)

```
signal_handler:
  store x.lo <- 0
  store x.hi <- 0
  return

f:
  load r.lo <- (x.lo)
  load r.hi <- (x.hi)
  compare r, 0
  branch-not-equal .out
  store x.lo <- 0xffffffff
  store x.hi <- 0xffffffff
.out:
  return
```

Signal processed here
= torn load

Signal processed here
= torn store

(Note: signal atomicity is not the same as concurrent read/write atomicity)

# Language problems

## Atomicity of loads and stores

An integer type that can be accessed atomically, for signal purposes

```
sig_atomic_t x;

void
signal_handler(int signo)
{
    x = 0;
}

void
f(void)
{
    if (x == 0)
        x = -1;
}
```

```
signal_handler:
  store x <- 0

  return

f:
  load r <- (x)

  compare r, 0
  branch-not-equal .out
  store x <- 0xffffffff

.out:
  return
```

(Note: signal atomicity is not the same as concurrent read/write atomicity)

# Language problems

## Reordering by the compiler

```
sig_atomic_t x;
sig_atomic_t y;

void
signal_handler(int signo)
{
    assert(x <= y);
}

void
f(void)
{
    x++;
    y++;
}
```

```
signal_handler:
  load r1 <- x
  load r2 <- y
  compare r1, r2
  branch-if-less-than-or-equal .out
  call assert_failed_abort
.out:
  return

f:
  load r <- y
  increment r
  store r -> y

  load r <- x
  increment r
  store r -> x

  return
```

Compiler decided to write to y first. A signal handled here sees the reordering.

# Language problems

## Reordering by the compiler

Volatile qualifier forces load/store order

```
volatile sig_atomic_t x;
volatile sig_atomic_t y;

void
signal_handler(int signo)
{
    assert(x <= y);
}

void
f(void)
{
    x++;
    y++;
}
```

```
signal_handler:
  load r1 <- x
  load r2 <- y
  compare r1, r2
  branch-if-less-than-or-equal .out
  call assert_failed_abort
.out:
  return

f:
  load r <- x
  increment r
  store r -> x

  load r <- y
  increment r
  store r -> y

  return
```

# (Non-problem) Out-of-order execution

## Modern architectures have *precise* interrupts

```
volatile sig_atomic_t x;
volatile sig_atomic_t y;

void
signal_handler(int signo)
{
    assert(x <= y);
}

void
f(void)
{
    x++;
    y++;
}
```

- Illusion of in-order serial execution is magically maintained while handling interrupts (eg by flushing pipeline, so interrupt doesn't have to wait for instructions to finish)
- (Note: this is independent of multi-threading/multi-processing problem, where you need explicitly memory barriers to control ordering!)

# Reentrancy

## Blowing the stack, or running non-reentrant code

```
void
signal_handler(int signo)
{
    do_something();
}

void
install_signal_handler(void)
{
    signal(SIGUSR1, signal_handler);
}
```

Old signal() interface doesn't block signals while handling them.

```
void
signal_handler(int signo)
{
    do_something();
}

void
install_signal_handler(void)
{
    struct sigaction sa = {
        .sa_handler = signal_handler;
    };
    sigaction(SIGUSR1, &sa, NULL);
}
```

sigaction() masks the given signal while already handling that signal (unless .sa_flags disables that)

# Deadlock

## We can’t use locks!

```
void
signal_handler(int signo)
{
    acquire_mutex(&m);
    count++;
    release_mutex(&m);
}

void
f(void)
{
    acquire_mutex(&m);
    count = 0;
    release_mutex(&m);
}
```

If the signal handler runs here, it will surely deadlock!

# Deadlock

## We can’t use atomics operations that might be emulated

```
void
signal_handler(int signo)
{
    pg_atomic_fetch_or_u64(&x, MY_FLAG);
}

void
f(void)
{
    pg_atomic_fetch_and_u64(&x, ~MY_FLAG);
}
```

This may be hiding a spinlock acquisition, on some platforms

# Unintented clobbering of state

```
void
signal_handler(int signo)
{
    write(some_fd, “.”, 1);
}

void
f(void)
{
    if (some_syscall() < 0)
    {
        if (errno == …)
        …
    }
}
```

If the signal handler runs here, write() might clobber errno!

# Unintented clobbering of state

```
void
signal_handler(int signo)
{
    int save_errno = errno;
    write(some_fd, “.”, 1);
    errno = save_errno;
}

void
f(void)
{
    if (some_syscall() < 0)
    {
        if (errno == …)
        …
    }
}
```

Value is restored

# Non-async-signal-safe functions

## Code that internally uses locks or modifies state

```
void
signal_handler(int signo)
{
    printf(“hello world\n”);  /* ! */
}

void
signal_handler(int signo)
{
    write(STDERR_FILENO, “hello world\n”, 12);
}
```

- POSIX gives a list of standard calls that are async-signal-safe; mainly:
  - Simple system calls, no user space state mutation
- Common mistakes
  - malloc(), printf(), exit(), …
  - palloc(), elog(), proc_exit(), …

# The system() call isn’t a system call

- After forking, but before executing a subprogram, a signal sent to a process group might be handled in parent *and* child

Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose.

- Henry Spencer, writing on comp.std.c
(about something else, I just love this quote)

# Synchronous signal handlers are different!

## (as long as they are only actually called synchronously)

```
/* signal handler for floating point exception */
void
FloatExceptionHandler(SIGNAL_ARGS)
{
    /* We're not returning, so no need to save errno */
    ereport(ERROR,
            (errcode(ERRCODE_FLOATING_POINT_EXCEPTION),
             errmsg("floating-point exception"),
             errdetail("An invalid floating-point operation was signaled. "
                       "This probably means an out-of-range result or an "
                       "invalid operation, such as division by zero.")));
}
```

Handler registered for the synchronous signal SIGFPE, but kill(1234, SIGFPE) would reach it asynchronously (!)

# Race to interrupt

## Missed it already?

```
vood
wait_for_interrupt(void)
{
    sleep(NAP_TIME);

    /* OR a common technique in older code for higher resolution timeout  */

    select(…, &timeout);
}
```

If the handler runs before we enter sleep(), we'll sleep

POSIX doesn't say whether select() returns with EINTR or restarts for SA_RESTART!

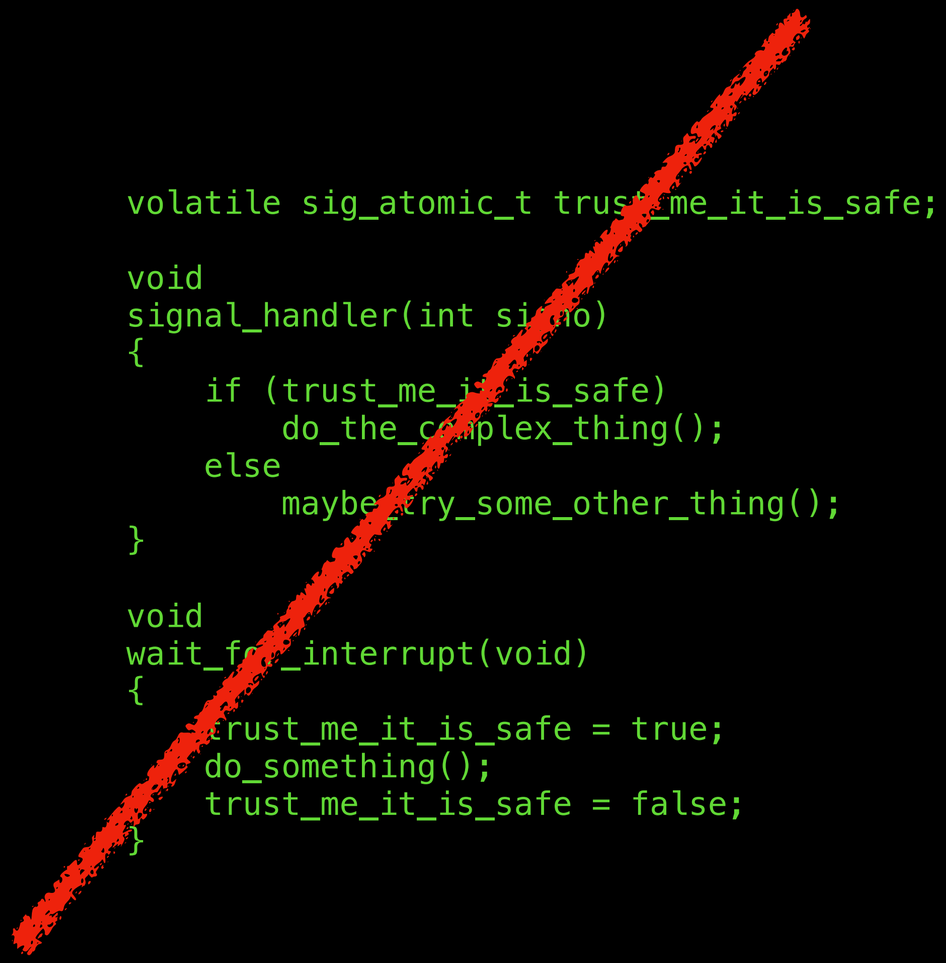
# Race to interrupt

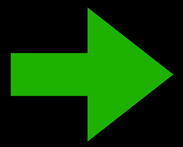
## Simple attempt to remember in a handler is still racy

```
volatile sig_atomic_t got_SIGINT;

void
SIGINT_handler(int signo)
{
    got_SIGINT = true;
}

vood
wait_for_interrupt(void)
{
    while (!got_SIGINT)
        sleep(NAP_TIME);
}
```

If the signal handler runs between these lines, we miss got_SIGINT but we enter sleep()!

```
volatile sig_atomic_t trust_me_it_is_safe;

void
signal_handler(int signo)
{
    if (trust_me_it_is_safe)
        do_the_complex_thing();
    else
        maybe_try_some_other_thing();
}

void
wait_for_interrupt(void)
{
    trust_me_it_is_safe = true;
    do_something();
    trust_me_it_is_safe = false;
}
```


volatile sig_atomic_t trust_me_it_is_safe;
void
signal_handler(int signo)
{
if (trust_me_it_is_safe)
do_the_complex_thing();
else
maybe_try_some_other_thing();
}
void
wait_for_interrupt(void)
{
trust_me_it_is_safe = true;
do_something();
trust_me_it_is_safe = false;
}



Part I: Signal handlers are dangerous
Part II: Modern PostgreSQL IPC APIs
Part III: Some ideas for future improvements

```
commit 2746e5f21d4dce07ee55c58b2035ff631470577f
Author: Heikki Linnakangas <heikki.linnakangas@iki.fi>
Date:   Sat Sep 11 15:48:04 2010 +0000

    Introduce latches. A latch is a boolean variable, with the capability to
    wait until it is set. Latches can be used to reliably wait until a signal
    arrives, which is hard otherwise because signals don't interrupt select()
    on some platforms, and even when they do, there's race conditions.

    On Unix, latches use the so called self-pipe trick under the covers to
    implement the sleep until the latch is set, without race conditions. On
    Windows, Windows events are used.

    Use the new latch abstraction to sleep in walsender, so that as soon as
    a transaction finishes, walsender is woken up to immediately send the WAL
    to the standby. This reduces the latency between master and standby, which
    is good.

    Preliminary work by Fujii Masao. The latch implementation is by me, with
    helpful comments from many people.
```

# Terminology hazard: “latch”

- In almost all database literature and RDBMSes (DB2, Oracle, SQL Server, MySQL, …), a latch means something like `pthread_mutex`
  - Copied from System/R or mainframe OS into other RDBMSs?</guess>
  - In PostgreSQL, we use the term LWLock for basic mutexes (lightweight lock, more soon)
- C++’s `std::latch` is something else again, like `pthread_barrier`
- PostgreSQL’s latch is more like a latch in electronics/ICs

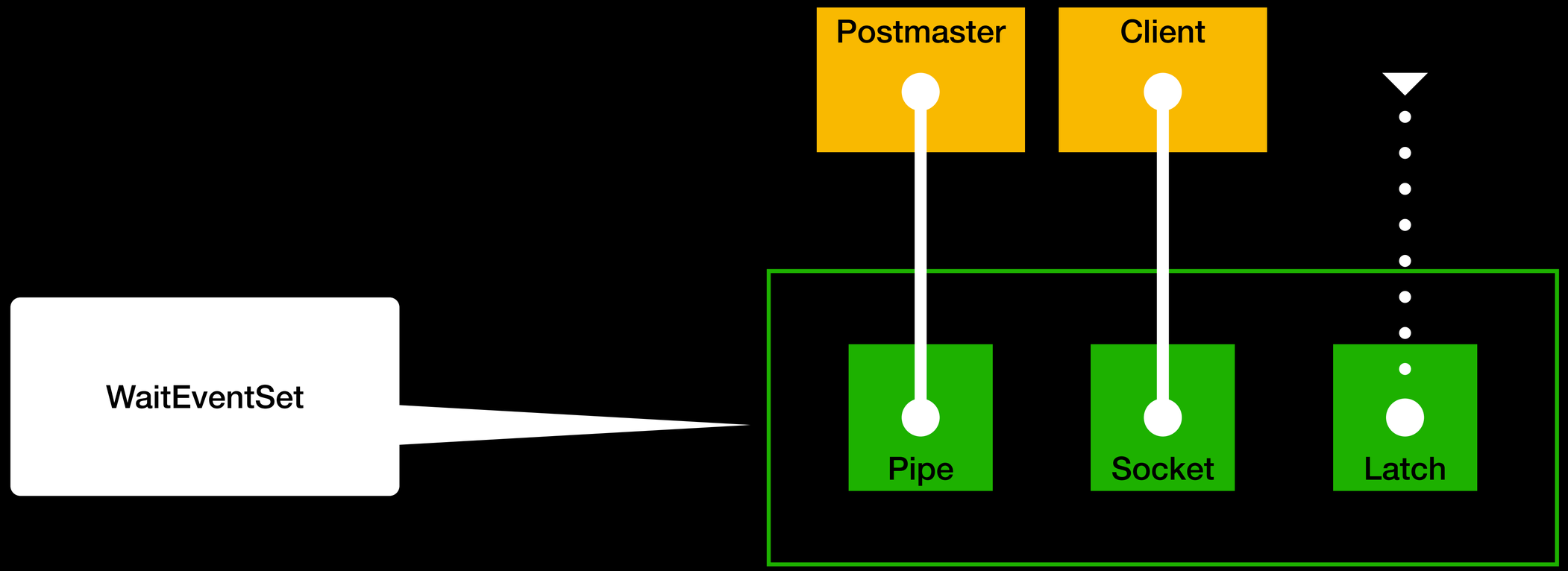
# Latches are multiplexable with sockets/pipes

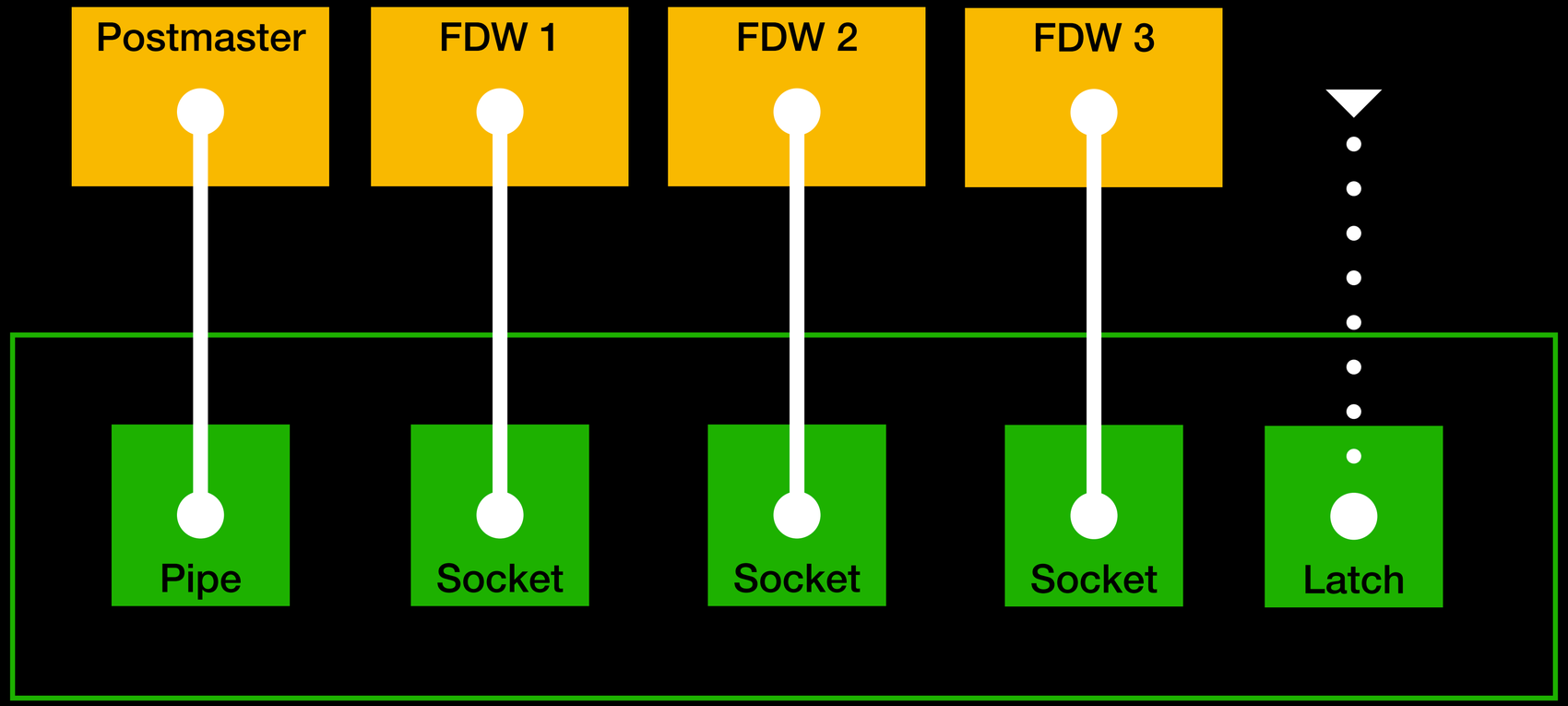
- If we only wanted to wait for a signal, and consume it synchronously, we could perhaps use sigwait() or sigtimedwait(), but then we couldn’t also wait for sockets and pipes at the same time
- We could instead have a pipe (or eventfd) for every backend, inherited by every backend, and then write(backend_pipes[n], "!”, 1) as a wakeup message, but that’d require a potentially huge number of descriptors
- With a signal we only need to know the PID of the recipient, and the receiver can multiplex a self-pipe, signalfd, or kqueue signal event

```
* There are three basic operations on a latch:
*
* SetLatch      - Sets the latch
* ResetLatch    - Clears the latch, allowing it to be set again
* WaitLatch     - Waits for the latch to become set
*
* WaitLatch includes a provision for timeouts (which should be avoided
* when possible, as they incur extra overhead) and a provision for
* postmaster child processes to wake up immediately on postmaster death.
* See latch.c for detailed specifications for the exported functions.
*
* The correct pattern to wait for event(s) is:
*
* for (;;)
* {
*     ResetLatch(<latch>);
*     if (work to do)
*         Do Stuff();
*     WaitLatch(<latch>, <events>, <timeout>, <wait_event_id>);
* }
*
* It's important to reset the latch *before* checking if there's work to
* do. Otherwise, if someone sets the latch between the check and the
* ResetLatch call, you will miss it and Wait will incorrectly block.
```

- ~~select()~~
- poll()
- epoll_wait()
- kevent()
- WaitForMultipleObjects()

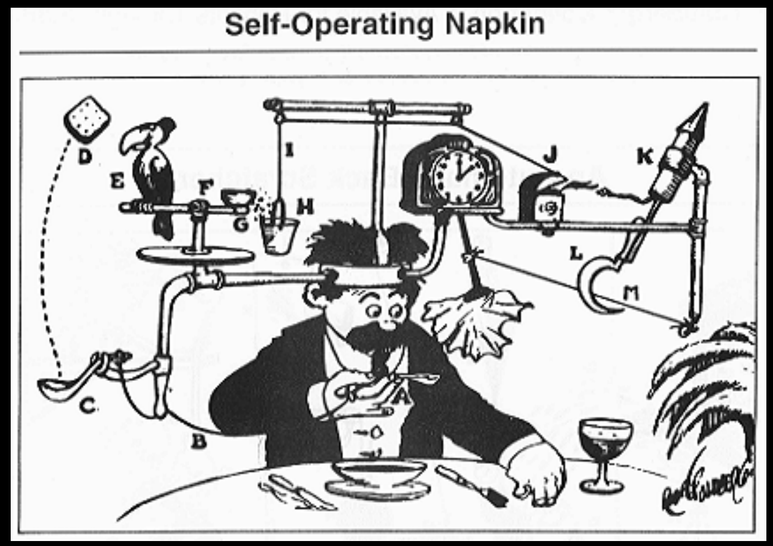
Stateful interfaces, avoid internal polling kernel objects (eg postmaster pipe) on every sleep




Postmaster
FDW 1
FDW 2
FDW 3
Pipe
Socket
Socket
Socket
Latch

# PMSignal

- Backends use “PMSignals” to ask the postmaster to do things.  The involves setting a shared memory flag eg PMSIGNAL_START_AUTOVAC_WORKER, and then sending SIGUSR1
- The postmaster’s SIGUSR1 handler just sets a flag and its own latch, to make its main loop return from WaitEventSetWait()
- We *could* just have the backend set the postmaster’s latch directly, and skip the handler.  (Robustness question.)



# ProcSignal

- When backends want to ask another backend to do certain things, they send ProcSignals, which work the same way: set a flag eg PROCSIG_LOG_MEMORY_CONTEXT and  send SIGUSR1
- The SIGUSR1 handler in most cases sets an “interrupt” flag, for the next call to CHECK_FOR_INTERRUPTS() to see and do something about*
- It also sets the backend’s latch, to break out of WaitEventSetWait() if we happen to be in it
- We *could* figure out how to skip SIGUSR1, and just set the latch directly from the sender, and teach CHECK_FOR_INTERRUPTS() to deal with the PROCSIG_XXX flags directly

*In some places we do more work than that directly in the SIGUSR1 handler, but that’s a bug to be fixed.

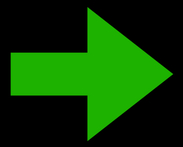
# ProcSignalBarrier

- PROCSIG_BARRIER asks every backend to do something
- The only current use of it is to force every backend to close *all* smgr file descriptors
  - Fixes random failures on Windows where you can’t unlink directories while someone has files open
  - Fixes historical bugs in hard cases where we lack invalidation, and could mix up files
- We are waiting for every backend in the system to reach CHECK_FOR_INTERRUPTS()!

# CHECK_FOR_INTERRUPTS()

## “CFI”

- Co-operation with the interrupt system is non-optional
  - Wait loops should do this when the latch is set
  - Long computations should figure out some place to put them too
- CHECK_FOR_INTERRUPTS() usually does nothing, but might throw ERROR, throw FATAL, or do some requested work and then return/continue
- Interrupts can be “held” with {HOLD,RESUME}_INTERRUPTS(). They are held automatically while any LWLock is held.
- Rarer case: {HOLD,RESUME}_CANCEL_INTERRUPTS(), suppresses only interrupts that would throw ERROR, used avoid protocol sync problems.
- Over the past decade, nearly everything that used to be done in signal handlers has been kicked out of there and into CHECK_FOR_INTERRUPTS()

Part I: Signal handlers are dangerous
Part II: Modern PostgreSQL IPC APIs

Part III: Some ideas for future improvements

# More fine-grained control of CFI()

- Problem: in some places we block interrupts, because we don’t want to ereport(ERROR); for example during a loop that cleans up temporary files on error
- That means we don’t handle ProcSignalBarrier code, for example

# Provide multiplexable subprocesses

- Problem: system() and popen() do not lend themselves to multiplexing
- It’s impossible to do portable non-blocking I/O with popen() because of FILE * interface
- We need to set up our own non-blocking pipes and use a WaitEventSet.  It’s not OK that COPY FROM PROGRAM does not process ProcSignalBarrier requests.

# Remove PMSignal and ProcSignal signals?

- Setting the target process's latch directly would be enough to wake it up if it's block in a wait loop
- Moving interrupt/ProcSignal flags into shared memory would let the CHECK_FOR_INTERRUPTS() see it, for compute-bound loops

# Do we still need to pretend that Windows has signals?

- Before we had higher level abstractions, it made more sense to port to Windows by emulating signals (incredible achievement)
- Would it be better if WaitEventSet had a first class way to consume process exits, that mapped to Window and Unix primitives?
- pg_ctl really opens a control pipe to talk to the server; why do we have to pretend it’s SIGQUIT etc?
- Likewise for pmsignals, which I already mentioned the idea of removing

fin