CS202 (003): Operating Systems
File System III, continued

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

Last Time

Journaling (borrowed from how transitions are implemented in databases)

Goal: Reduce write/space overhead without violating atomicity

Treat file system operations as transactions:
after a crash, failure recovery ensures

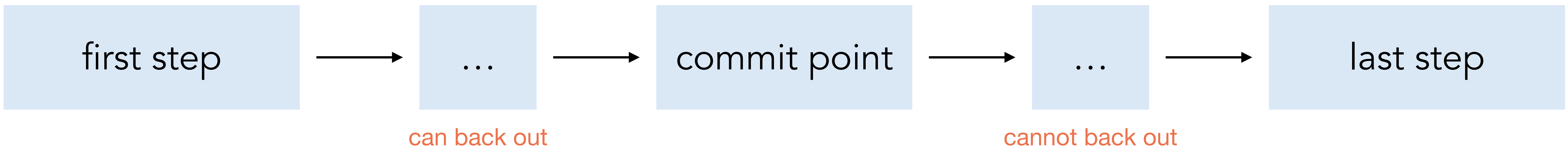
1. committed file system operations are reflected in on-disk data structures
2. uncommitted file system operations are not visible after crash recovery

Record enough information to finish applying committed operations *(redo operations)* and/or roll-back uncommitted operations *(undo operations)*
This information is stored in a redo/undo log

Journaling

Commit point: the point at which there is no turning back

for example



What is the commit point in copy-on-write?

Journaling — redo logging (used by ext3 & ext4)

FS computes what would change due to a operation

↓

FS computes where in the log it can write this transaction, and writes a transactions begin record . Do not have to wait on this.

↓

FS writes a record(s) with all the changes it computed in step 1.
FS **must wait** for changes and TxnBegin to finish written to the disk

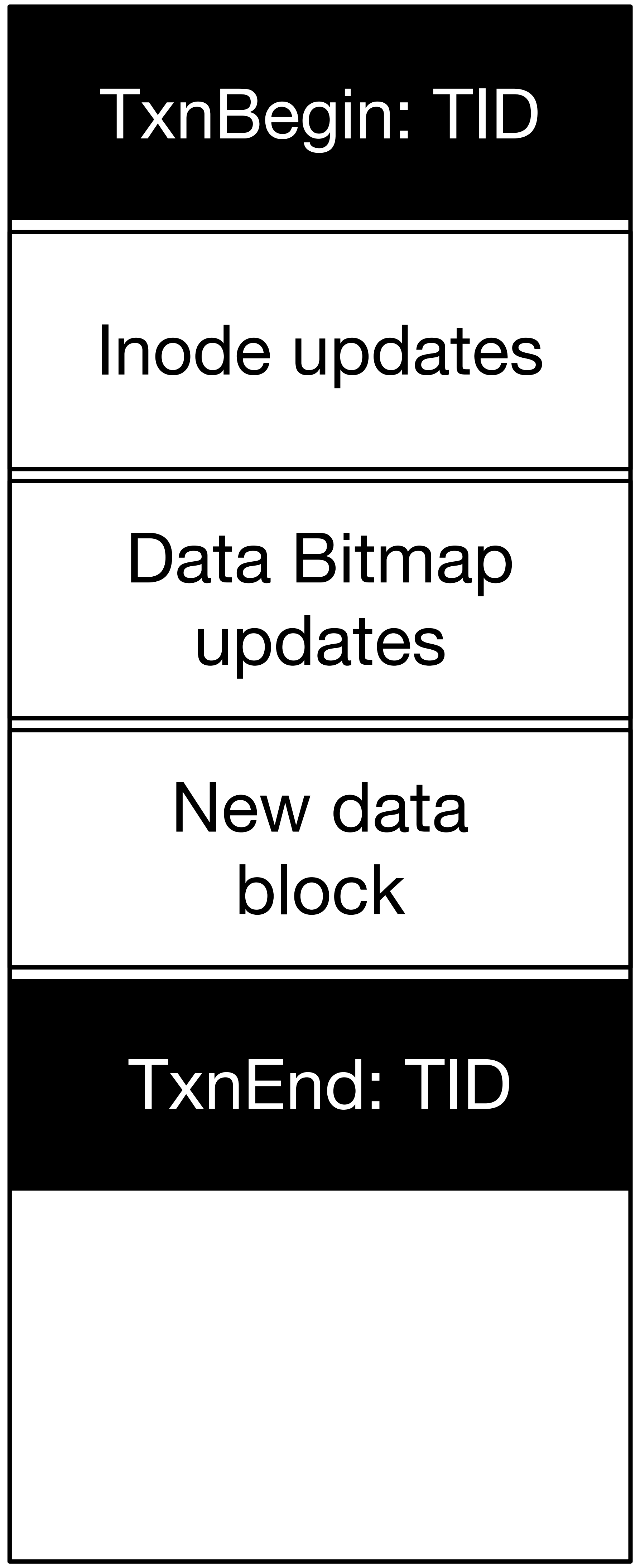
↓

Once step 3 finishes, the system writes a transaction end record

↓

Once the TxnEnd has been written, the FS asynchronously performs the actual FS changes

“checkpointing”



ext3 journal layout

Journaling — crash recovery of redo logging

High-level idea:
read through the logs, find **committed operations** and apply them

How to check whether ops are committed? Look at TxnBegin and TxnEnd!
It is safe to apply the same redo log multiple times

FS starts scanning from the **beginning of the log**

↓

Every time it finds a TxnBegin entry, it looks for the corresponding TxnEnd entry

↓

If matching (TxnBegin, TxnEnd) found, FS checkpoints the changes

↓

Recovery is completed once the entire log is scanned

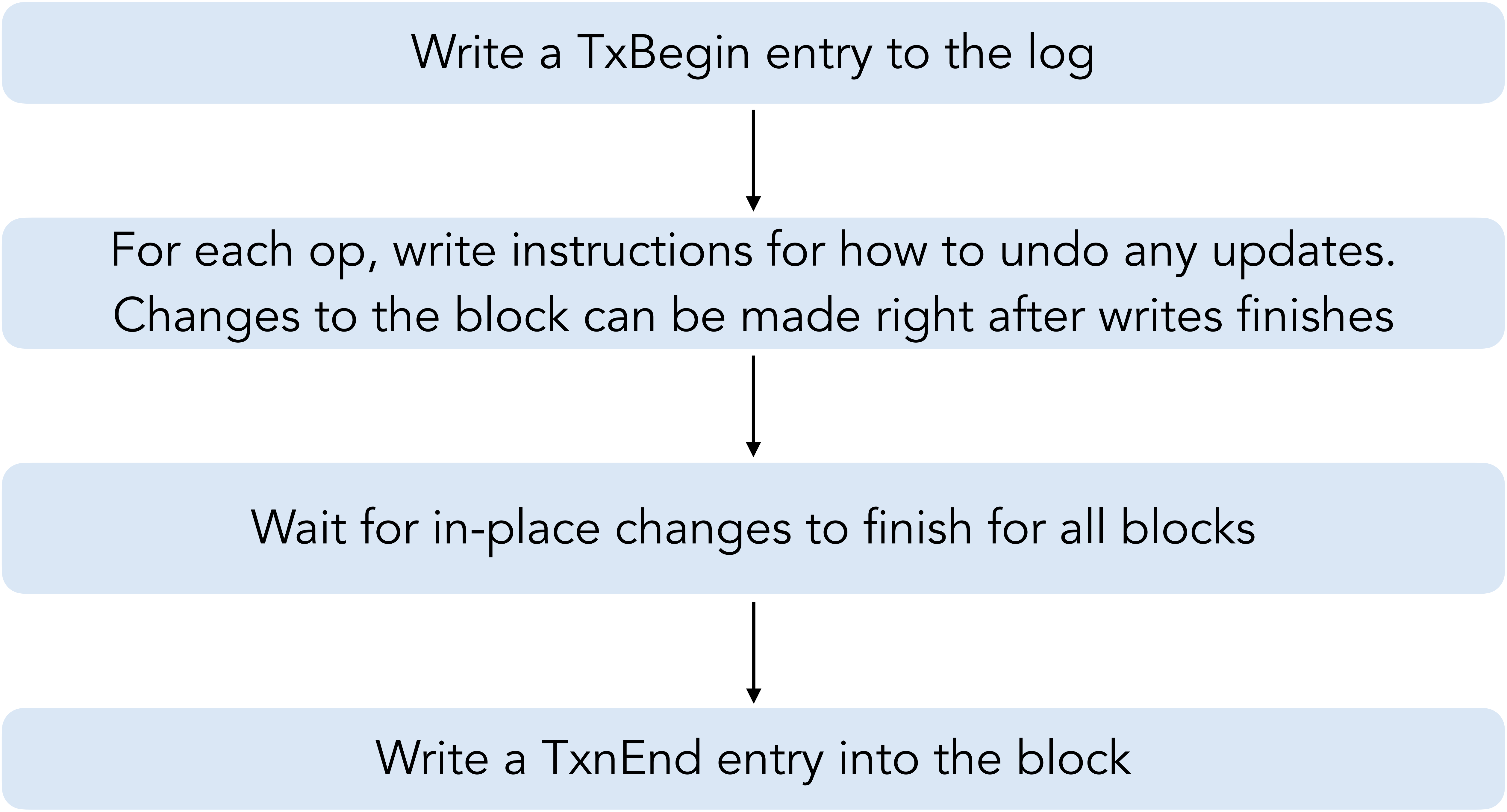
What to log?

Logging can double the amount of data written to the disk
Ext3 and 4 allows user to choose what to log

Default: metadata only (assuming people are fine with data loss after crash)
Can change to force data to be logged w/ metadata

Journaling — undo logging

(Not used in isolation by any file system)



all changes have been written to the actual FS data structures

Journaling — crash recovery from undo logging

Scan to find all uncommitted transactions from **the end of the log**

↓

For each such transaction, check whether undo entry is valid (checksum)

↓

Apply all valid undo entries found

disk back to a consistent state

Benefits

Changes can be checkpoints to disk as soon as the undo log has been updated — useful when the amount of buffer cache is low

Disadvantages

A transaction is not committed until all dirty blocks have been flushed to their in-place targets

Redo logging vs. Undo logging

Benefits	A transaction can commit without all in-place updates (writes to actual disk locations) being completed — useful when in-place updates might be scattered all over the disk
Disadvantages	A transaction's dirty blocks need to be kept in the buffer-cache until the transaction commits and all of the associated journal entries have been flushed to disk. This might increase memory pressure.

Benefits	Changes can be checkpoints to disk as soon as the undo log has been updated — useful when the amount of buffer cache is low
Disadvantages	A transaction is not committed until all dirty blocks have been flushed to their in-place targets

Combining Redo/Undo Logging (Done by NFTS)

Goal: allow dirty buffers to be flushed as soon as their associated journal entries are written. Transactions are committed as soon as logging is done

Reduce memory pressure when necessary, and have greater flexibility when scheduling disk writes

FS computes what would change due to a operation

↓

FS computes where in the log it can write this transaction, and writes a transactions begin record . Do not have to wait on this.

↓

FS writes both a redo log entry and an undo log entry for each of the changes computed in Step 1.
In-place changes can be made once the log information is written.

↓

Once TxnBegin and logs are written, write a TxnEnd entry

↓

Once the TxnEnd has been written, the FS asynchronously performs the actual FS changes

Journaling — crash recovery from redo+undo logging

Step 1: Redo pass

FS starts scanning from the **beginning of the log**

↓

Every time it finds a TxnBegin entry, it looks for the corresponding TxnEnd entry

↓

If matching (TxnBegin, TxnEnd) found, FS checkpoints the changes

↓

Recovery is completed once the entire log is scanned

Step 2: Undo pass

Scan to find all uncommitted transactions from **the end of the log**

↓

For each such transaction, check whether undo entry is valid (checksum)

↓

Apply all valid undo entries found

disk back to a consistent state

Designed for a time when the same Operating System ran on machines with very little memory (8-32MB), and also on "big-iron" servers with lots of memory (1GB+).
This was an attempt to get the best of both worlds.

CS202 (003): Operating Systems
How Debugger Works

Instructor: Jocelyn Chen

Most of the materials covered in this slide come from the lecture notes of Mike Walfish's CS202

What is a debugger?

A process that has some control over another process

“target”
“process being debugged”

Why is debugger cool?

The high-level functionality is invaluable to software developers

Set breakpoints
(`break`)
Pause a process
(`attach`)
Single step through the process
(`stepi` is one assembly instruction, `step` is one line of C code)
Continue process execution from the paused points
(`continue`)
Generate a stack trace
(`backtrace`)
Read and modify values of variables, which might be on the stack, heap or data segment
(`x` or `print` to read, `set $varname` to modify)
Read and modify program code (the TEXT area)
(`disassemble` to read)
Read and modify program registers
(`info registers` and `print` to read, and `set $regname` to write)
Modify parameters and return values for system calls
(`call` and `catch syscall`)
Set watchpoints
(`watch`, `awatch` and `rwatch`)

Why is debugger cool?

We have talked about that “Process are isolated from each other”

How can debugger access the target’s memory?

How can debugger stop target’s execution at specific address?

How can debugger “single-step” another process?

Why is debugger cool?

Debugger requires a lot of effort to make it work!

Stack frames

Virtual memory

Interrupts

Signals

Operating system and CPU

Attach and controlling a process

Launch a process that is attached

```
void launch_attached(const char* path,
                     char* const argv[]) {
    int pid = fork();
    if (pid == 0) {
    ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    execv(path, argv);
    }
    return pid;
}
```

Attach to a running process

```
void attach_to_process(pid_t pid) {
    ptrace(PTRACE_ATTACH, pid, NULL, NULL);
}
```

https://man7.org/linux/man-pages/man2/ptrace.2.html

How the debugger “synchronizes” with a process?

```
void continue_once_attached(pid_t pid) {
    while (1) {
        int status;
        waitpid(pid, &status);
        if (WIFSTOPPED(status)) {
            // The reason for the change was that pid stopped.
            // We should have stopped because of either SIGTRAP and SIGSTOP.
            assert(WSTOPSIG(status) == SIGTRAP || WSTOPSIG(status) == SIGSTOP);

            // Continue execution
            ptrace(PTRACE_CONT, pid, NULL, NULL);
            break;
        } else if (WIFEXITED(status)) {
            // The process exited before we could attach.
            printf("Process exited\n");
            break;
        }
    }
}
```

How the debugger stops a process?

```
void interrupt_target(pid_t pid) {
    // kill() is a system call that sends OS signals
    kill(pid, SIGSTOP);
    // Must use waitpid in order to wait for the signal to be delivered.
}
```

How the debugger reads/writes memory and registers?

```c
// Execute a single instruction in the process.
ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);

// Get non-floating point registers.
// This includes rsp, rip, rbp, etc.
struct user_regs_struct regs;
ptrace(PTRACE_GETREGS, pid, &regs, NULL);

// Get floating point registers.
struct user_fpregs_struct fpregs;
ptrace(PTRACE_GETFPREGS, pid, &fpregs, NULL);

// Set registers. This can be used to update
// register values.
ptrace(PTRACE_SETREGS, pid, &regs, NULL);
```

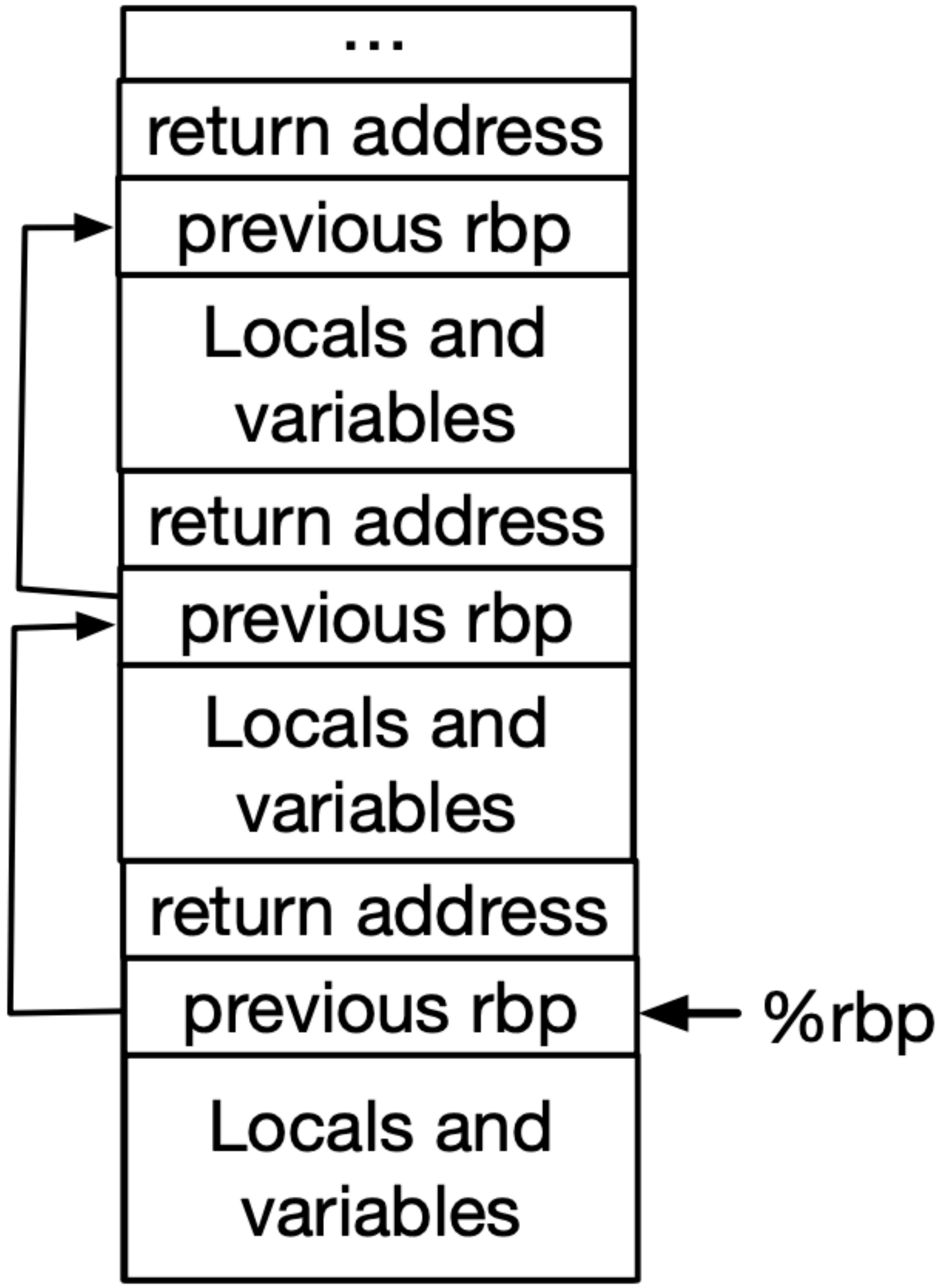
```c
// Note: PTRACE_PEEKUSER and PTRACE_POKEUSER
// provide a more efficient way to read or
// write a single register.

// Read a word (8 bytes) from address `addr`
// in target process memory. Note, despite being
// called PTRACE_PEEKDATA, on Linux this can
// read any part of memory, including the
// text segment.
uint64_t val;
val = ptrace(PTRACE_PEEKDATA, pid, addr, NULL);

// Write a word (8 byte) to address `addr` in
// target procss memory.
ptrace(PTRACE_POKEDATA, pid, addr, val);

// Get information on the signal that caused
// the target procss to stop.
siginfo_t sinfo;
ptrace(PTRACE_GETSIGINFO, pid, &sinfo, NULL);
```

How to get stack traces?




```
PTRACE_GETREGS → %rbp

return_addr = *(%rbp + 8)
```

Using it, the debugger can lookup:

- which function we just came from
- which line in the source

```
prev_rbp = *%rbp
```

%rbp contains a pointer to the previous frame's %rbp

```
%rbp → previous %rbp → previous %rbp → ...
```

Eventually it reaches:

```
%rbp = 0
```

This unwinds the stack frame-by-frame

```
current function
  → its caller
→ caller's caller
   → ...
  → __start
```

Stack trace!

How to get actual function names and line numbers?

So far, we are only talking about addresses
We need meta-data!

Key: Symbol tables and symbol files

Address => Global variable names
Address => Function names
Address => Source file names and line numbers

Symbols are best efforts, and in practice debuggers cannot always resolve names to values due to compiler optimizations!

Single Stepping

Key: Rely on hardware to do this!

```
OS sets TF = 1 in RFLAGS
CPU executes 1 instruction
CPU raises INT 1 (debug interrupt)
Kernel stops the process and reports SIGTRAP
Debugger returns control to user
```

Breakpoint

Naive Implementation

```
loop:
    single-step
    read RIP
    if RIP == breakpoint_addr: stop
```

Really Slow

Better-performing Solution

```
orig = PEEKDATA(addr)
save orig
POKEDATA(addr, 0xCC)   // write int 3
continue
```

Setting a breakpoint

```
CPU executes 0xCC
→ INT 3
→ kernel → SIGTRAP
→ target process stops
→ debugger wakes via waitpid()
```

Hitting a breakpoint

```
POKEDATA(addr, orig)      // restore true instruction
SINGLESTEP()              // execute it exactly once
POKEDATA(addr, 0xCC)      // reinsert breakpoint
```

Continuing after breakpoint

Watchpoints

Watchpoints stop execution whenever the specified memory address is read (`rwatch`) or accessed (`awatch`)

3 ways to implement

1. Single-step method   Slow, fallback
2. Page-fault method   Easy, coarse (page-sized)
3. Hardware method   Fast, but only 4 slots (DR0–DR3)

Page-fault method

the debugger asks the kernel to mark a page in the process as inaccessible

Hardware method

The processor will generate an interrupt whenever the program accesses one these addresses