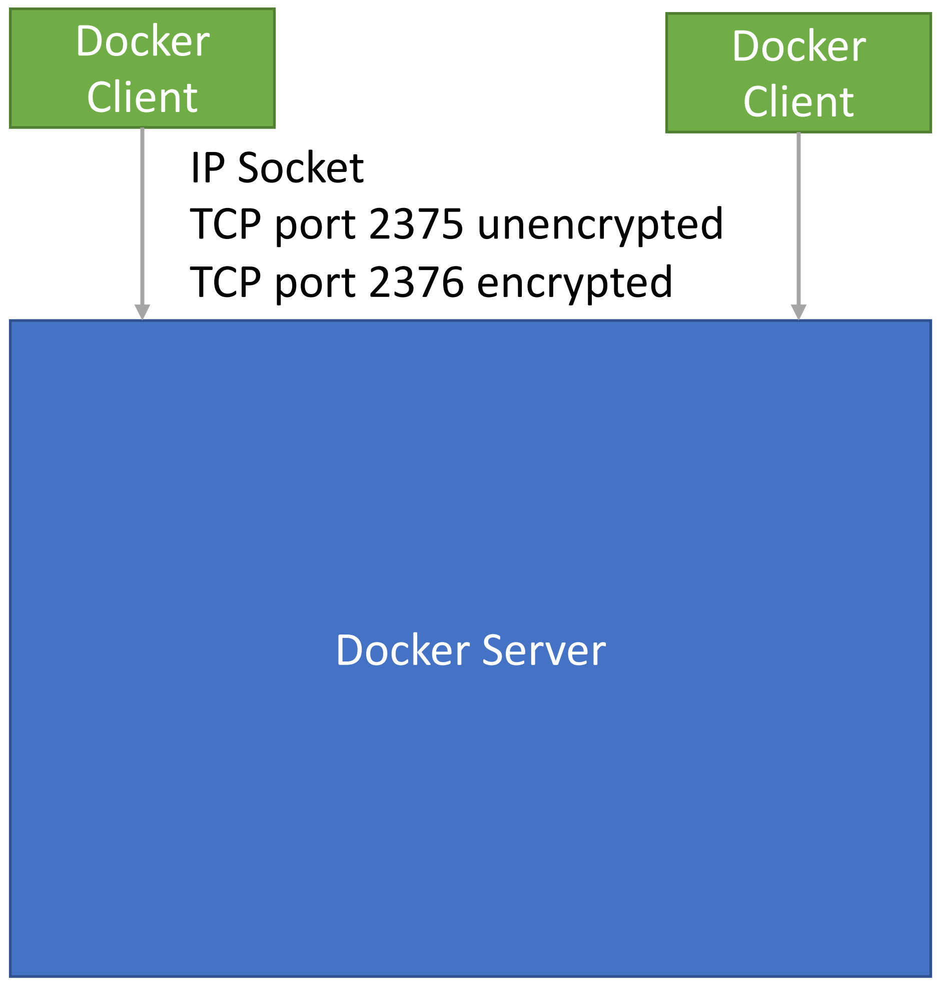
# Resource Virtualization with Containers

Tanu Malik

School of Computing, DePaul University

Visiting Faculty, CSE, IIT, Delhi

# Docker Architecture



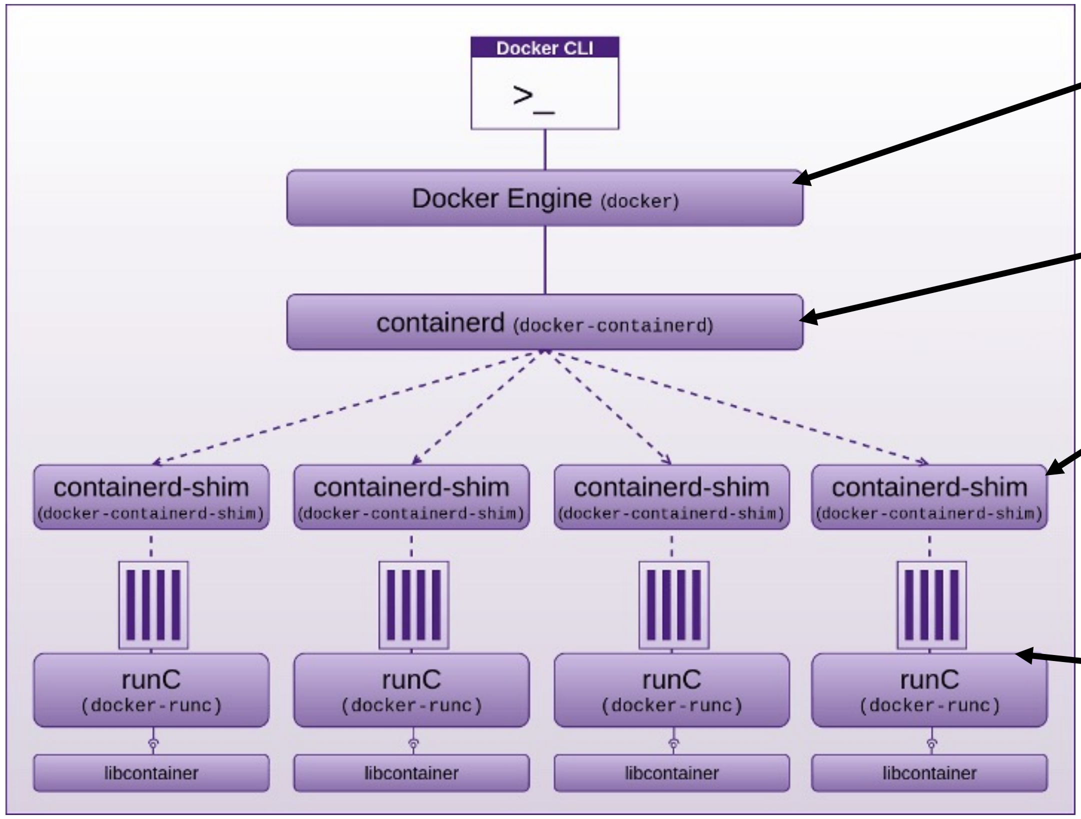
- Client/Server architecture
- Uses IP sockets for communication
- Clients can be on the same machine, or communicate over a network
- Allows connections from multiple concurrent clients!

# Docker Archtiecture

- Provides API to clients
- manages images, volumes, and builds

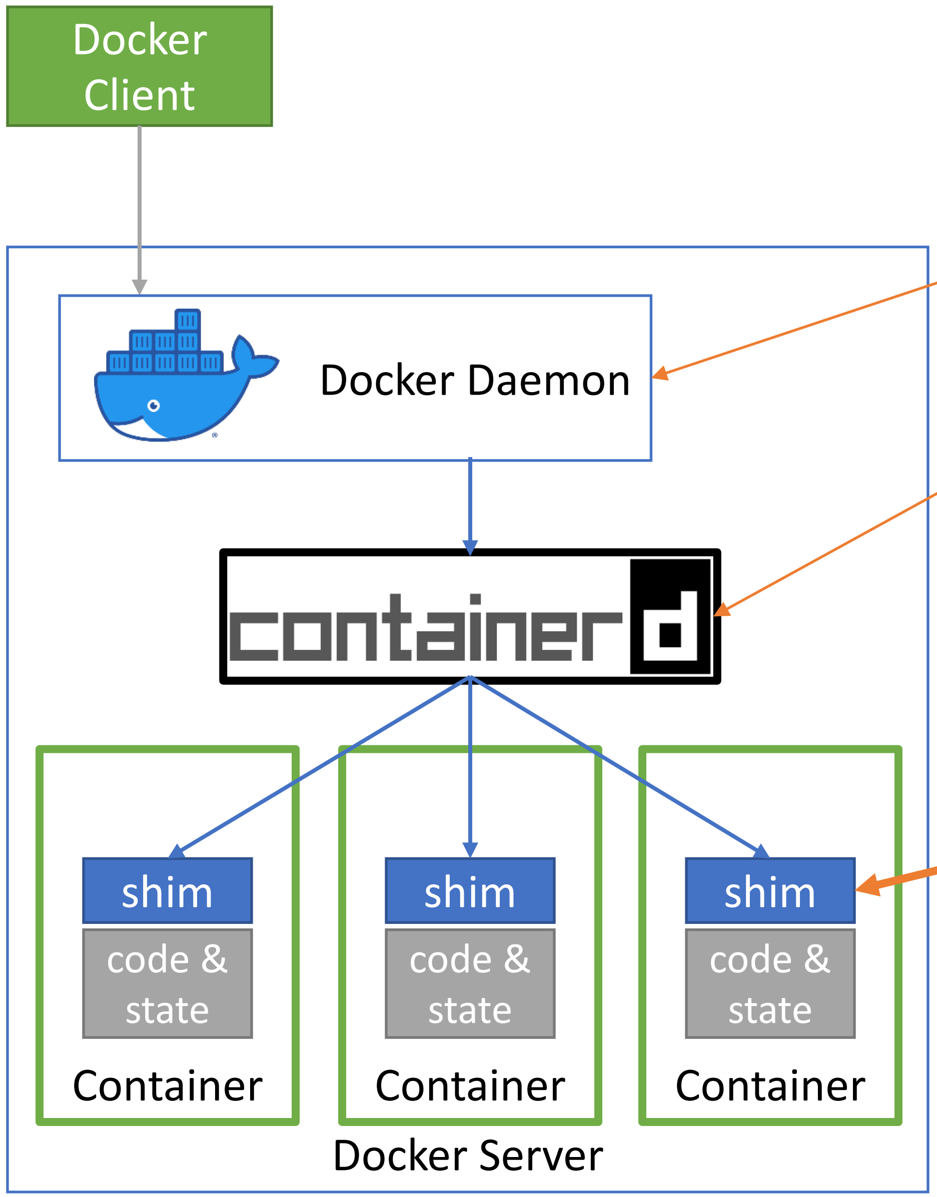
- Starts new containers with **runc**
- Supervises and controls state of running containers, i.e., tasks like starting, stopping, pausing or destroying containers

- Shim becomes PID=1 in the container.


Docker CLI
>_
Docker Engine (docker)
containerd (docker-containerd)
containerd-shim (docker-containerd-shim)
containerd-shim (docker-containerd-shim)
containerd-shim (docker-containerd-shim)
containerd-shim (docker-containerd-shim)
runC (docker-runc)
runC (docker-runc)
runC (docker-runc)
runC (docker-runc)
libcontainer
libcontainer
libcontainer
libcontainer

- **runc** reads container image specification
- Initializes container environment: namespaces, cgroups, network (covered later), etc.
- Creates persistent shim component
- Loads container programs and state
- **runc** exits when initialization is complete

Containers belong to containerd

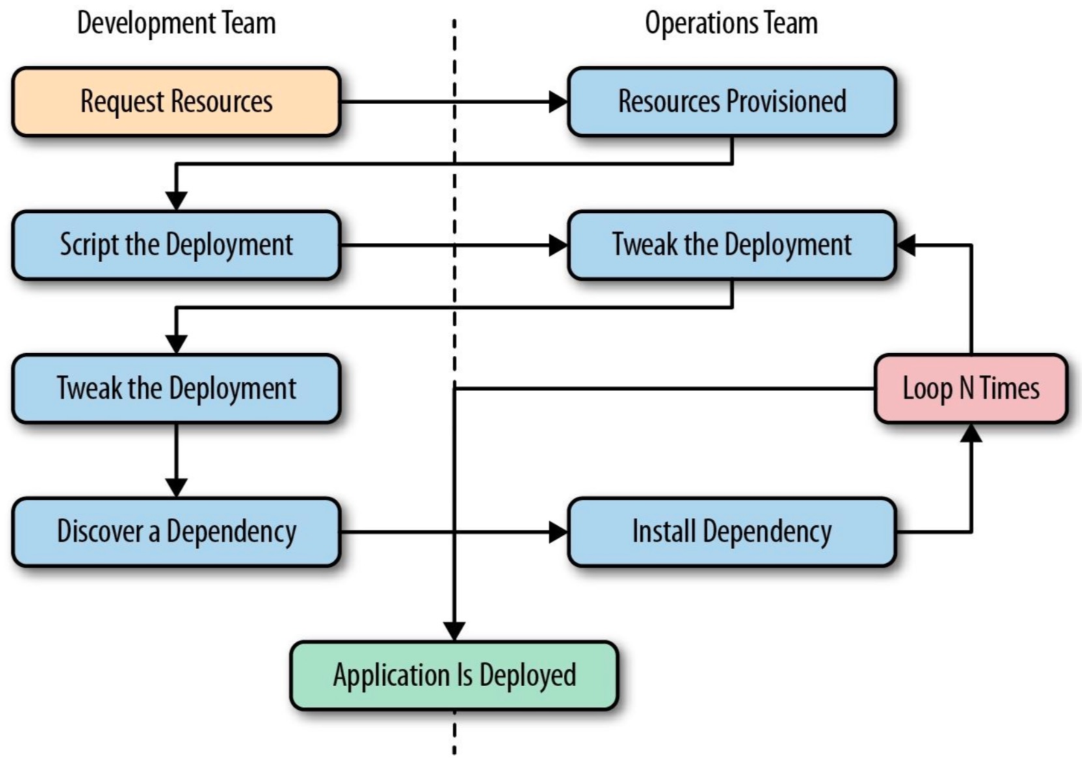


Provides API to clients

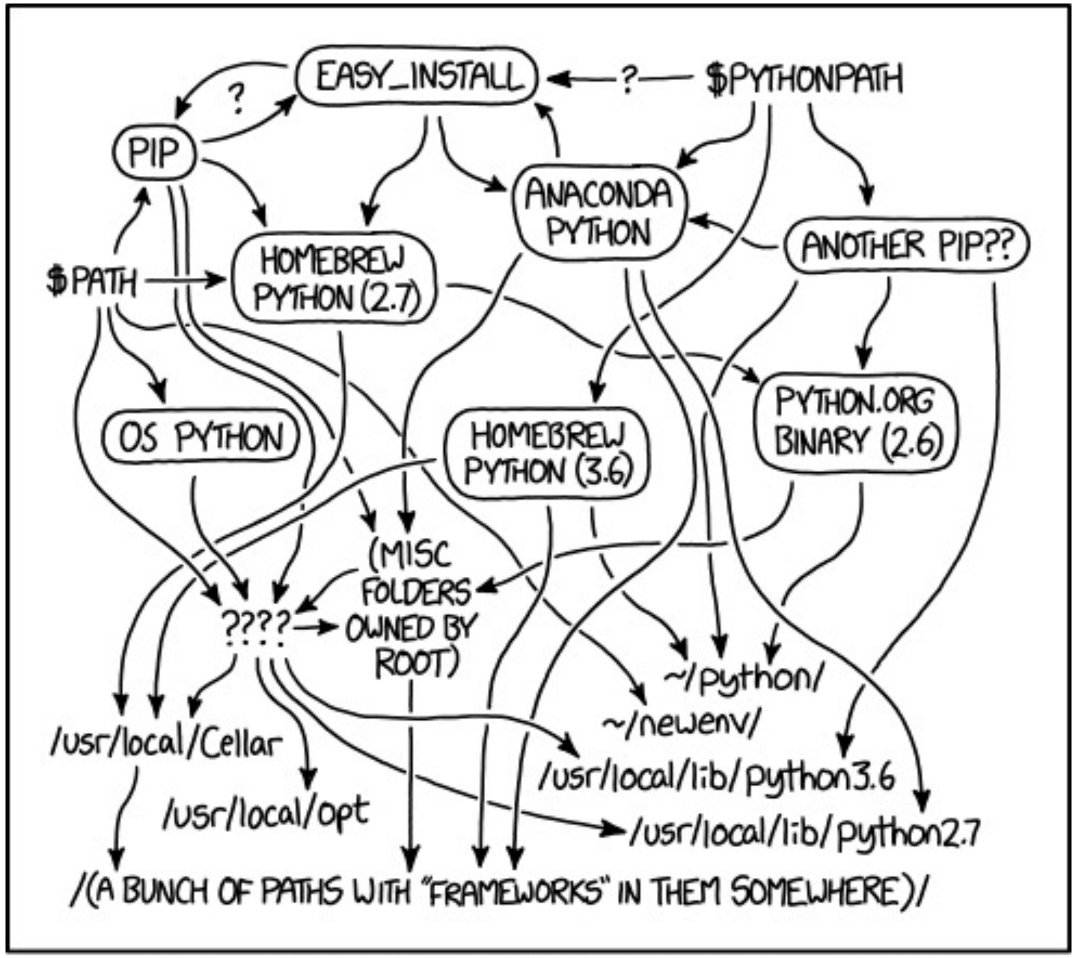
Supervises and controls state of running containers

- **containerd** clones a new **runc** for every container it creates
- Resource intensive to maintain a copy of **runc** for each container
- **shim** process maintains minimal environment to allow container to maintain communication with **containerd**
- This modular architecture allows containers to stay online even if the Daemon is restarted/upgraded!

# A traditional deployment workflow

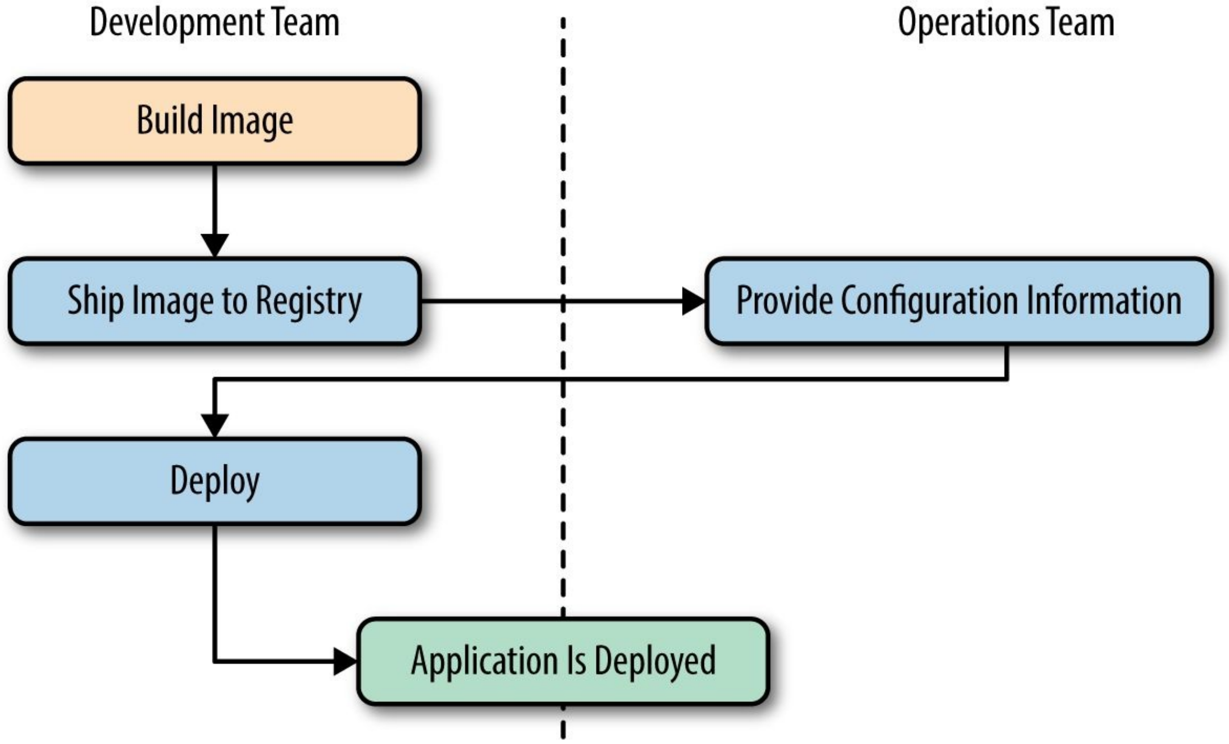


# Dependency Hell



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

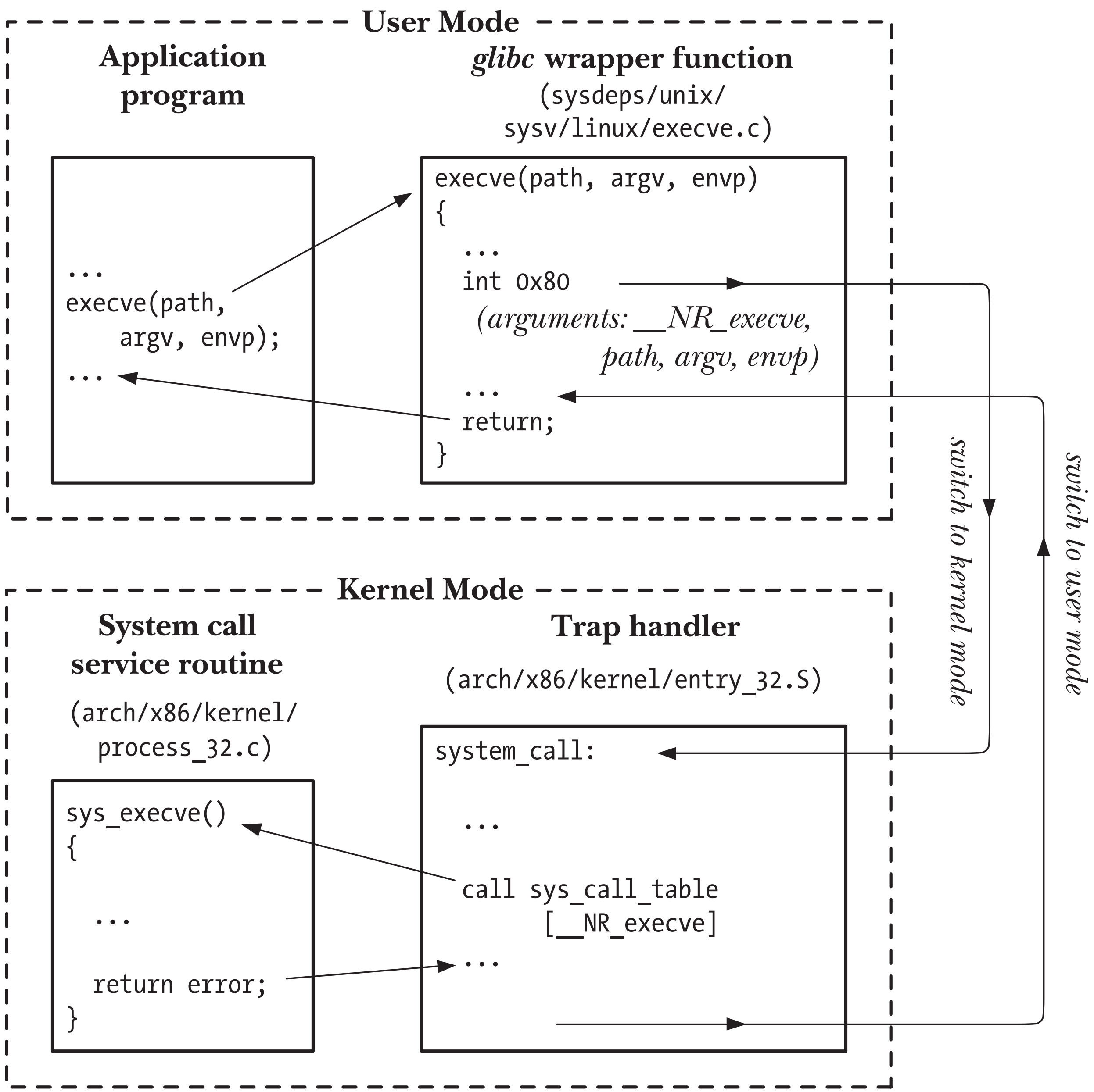
## Docker approach



## Sciunit

- Sciunits are efficient, lightweight, self-contained packages of computational experiments that can be guaranteed to repeat or reproduce regardless of deployment issues.

Idea: Audit + Copy + Redirect w/ PID isolation
= Automatic containerization


User Mode
Application program
glibc wrapper function
(sysdeps/unix/sysv/linux/execve.c)
...
execve(path, argv, envp);
...
execve(path, argv, envp)
{
...
int 0x80
(arguments: __NR_execve, path, argv, envp)
...
return;
}
switch to kernel mode
switch to user mode
Kernel Mode
System call service routine
(arch/x86/kernel/process_32.c)
Trap handler
(arch/x86/kernel/entry_32.S)
sys_execve()
{
...
return error;
}
system_call:
...
call sys_call_table [__NR_execve]
...

## ptrace

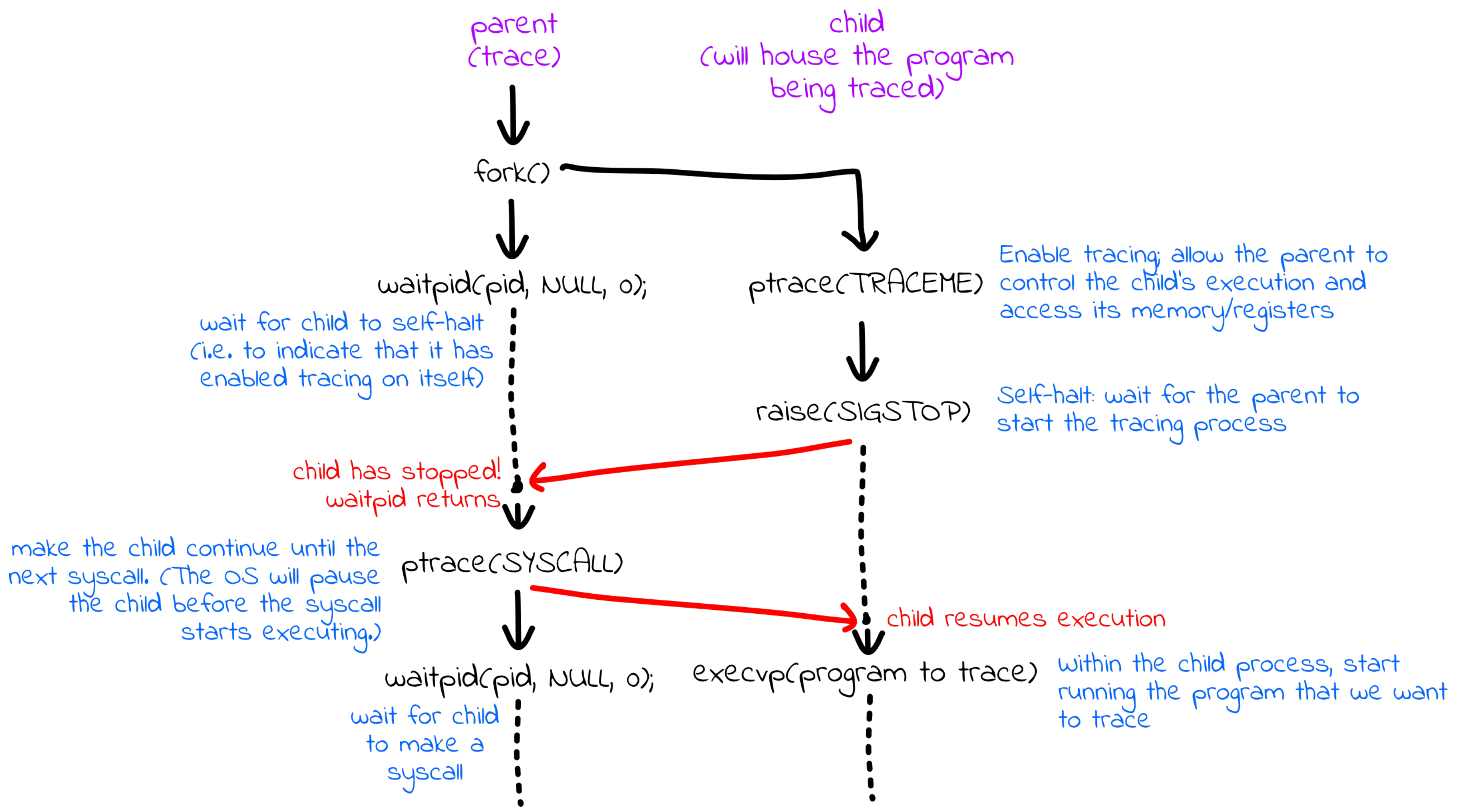
- The ptrace system call provides a means by which one process (the “tracer”) may observe and control the execution of another process (the “tracee”), and examine and change the tracee’s memory and registers. It is primarily used to implement breakpoint debugging and system call tracing
- Really, really, complicated syscall that can do a lot of things
- Basis for GDB

## ptrace

- long ptrace(enum __ptrace_request request, ... other arguments ...);

- Request is a constant indicating what you want to do:
  - PTRACE_TRACEME,
  - PTRACE_PEEKUSER,
  - PTRACE_SYSCAL
- Based on this constant other arguments might be required.

## Starting to trace


parent
(trace)
child
(will house the program being traced)
fork()
waitpid(pid, NULL, 0);
ptrace(TRACEME)
Enable tracing; allow the parent to control the child's execution and access its memory/registers
wait for child to self-halt (i.e. to indicate that it has enabled tracing on itself)
raise(SIGSTOP)
Self-halt: wait for the parent to start the tracing process
child has stopped! waitpid returns
make the child continue until the next syscall. (The OS will pause the child before the syscall starts executing.)
ptrace(SYSCALL)
child resumes execution
waitpid(pid, NULL, 0);
execvp(program to trace)
within the child process, start running the program that we want to trace
wait for child to make a syscall

- The child process calls ptrace(PTRACE_TRACEME) to inform the OS that its content is being monitored by the parent process

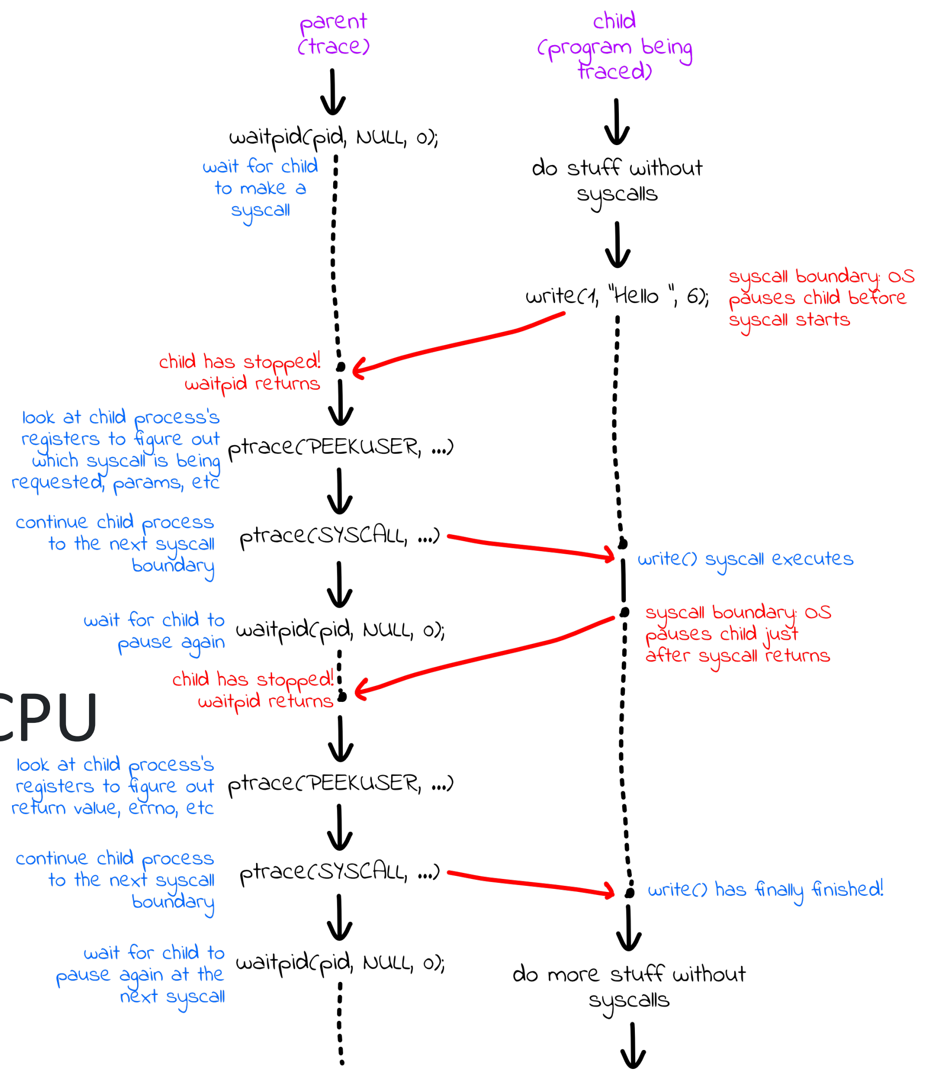
# ptrace

- ptrace(PTRACE_TRACEME) should be called in the child process, before starting to execute the program we want to trace, to tell the kernel that it wants its parent process to be able to observe its execution.
    - When the tracing starts, the child process will be paused (as in SIGSTOP paused; it will not continue until it receives SIGCONT).
- ptrace(PTRACE_SYSCALL, pid_t pid, 0, 0) should be called in the parent process, which will send SIGCONT to the child to wake it up,
    - PTRACE_SYSCALL will pause the child right when it asks the kernel to do something, and right when the kernel finishes running the syscall.
    - i.e., will pause the child again as soon as it reaches a syscall boundary.
- ptrace(PTRACE_PEEKUSER, pid_t pid, REGISTER_NAME * sizeof(long)) is called in the parent process in order to read a register value from inside the child process.
- ptrace(PTRACE_PEEKDATA, pid_t pid, void *address_to_read) is called by the parent to read a long (8 bytes) from address_to_read in the child’s virtual address space.

# Stopping before and after the system call

- Ptracing happens by
  Freezing child process,
  extracting data,
  restarting child process, and
  Repeating it

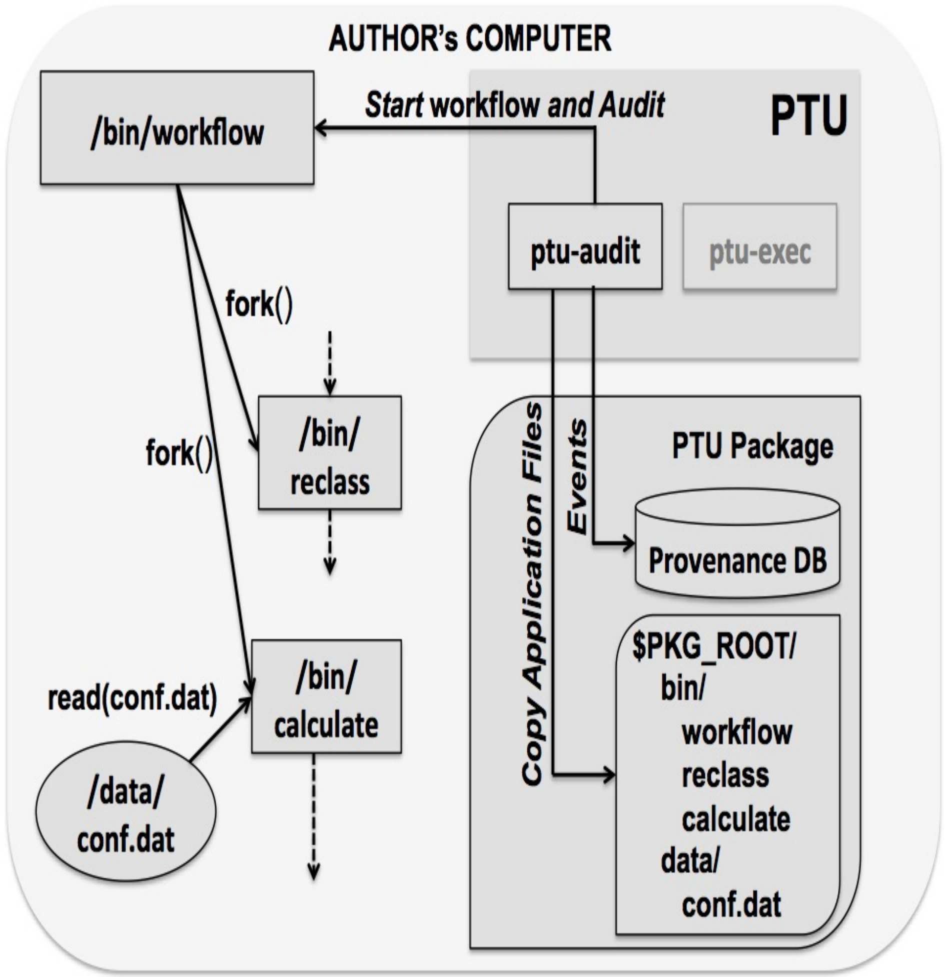
- The parent accumulates a series of CPU snapshots as the tracee oscillates in and out of its system calls.


parent (trace)
child (program being traced)
waitpid(pid, NULL, 0);
wait for child to make a syscall
do stuff without syscalls
write(1, "Hello ", 6);
syscall boundary: OS pauses child before syscall starts
child has stopped! waitpid returns
look at child process's registers to figure out which syscall is being requested, params, etc
ptrace(PEEKUSER, ...)
continue child process to the next syscall boundary
ptrace(SYSCALL, ...)
write() syscall executes
wait for child to pause again
waitpid(pid, NULL, 0);
syscall boundary: OS pauses child just after syscall returns
child has stopped! waitpid returns
look at child process's registers to figure out return value, errno, etc
ptrace(PEEKUSER, ...)
continue child process to the next syscall boundary
ptrace(SYSCALL, ...)
write() has finally finished!
wait for child to pause again at the next syscall
waitpid(pid, NULL, 0);
do more stuff without syscalls

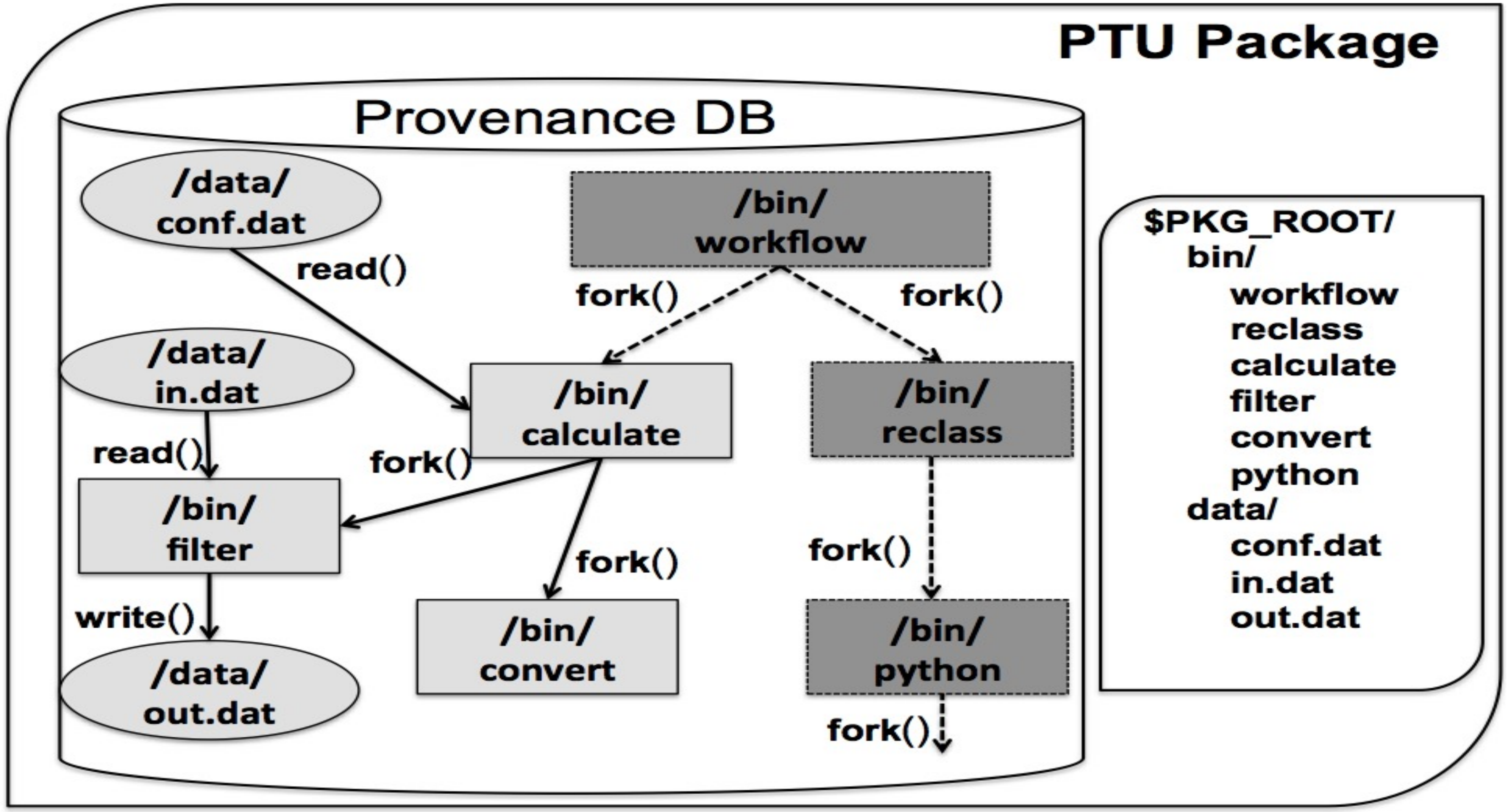
## Stopping before and after the system call

- The kernel will pause the child just as it's about to make a syscall, and it will wake up the parent. The parent uses PTRACE_PEEKUSER and PTRACE_PEEKDATA to look at the child process's registers and memory, figuring out which system call is being executed and what arguments were provided.
- The parent uses PTRACE_SYSCALL again to run the child until the next syscall boundary (which is when the syscall is returning).
- The kernel will pause the child again just as it's returning from the syscall, and it will wake up the parent. The parent uses PTRACE_PEEKUSER and PTRACE_PEEKDATA again to get the return value of the syscall, and to read errno from the child's memory if the syscall returned -1.
- The parent calls PTRACE_SYSCALL to wake up the child again, and waits for the child to make its next syscall.
-

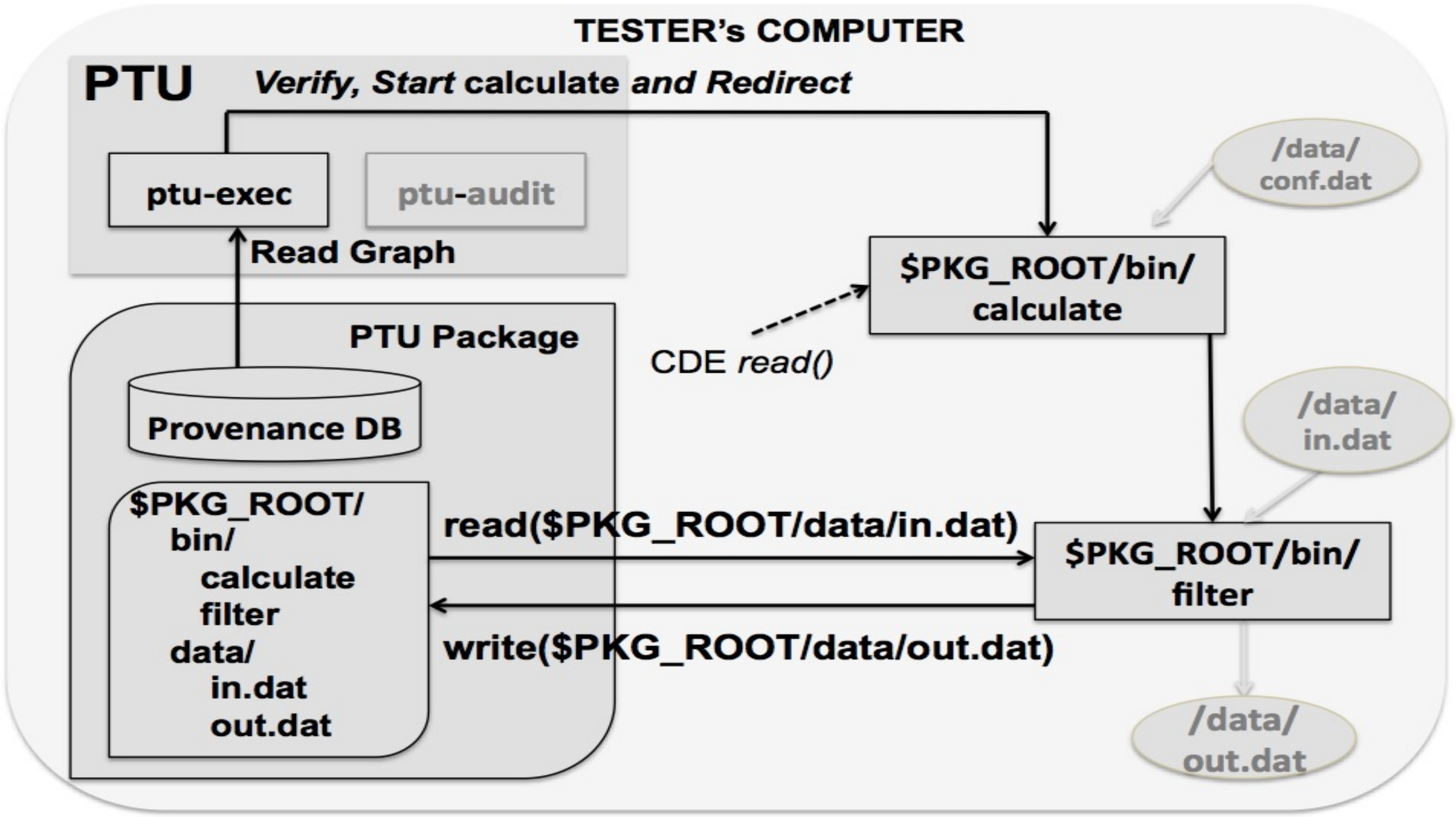
- Uses *ptrace* to monitor system calls
  - execve, sys_fork
  - read, write, sys_io
  - bind, connect, socket
- Collects provenance
- Collects runtime information
- Makes package



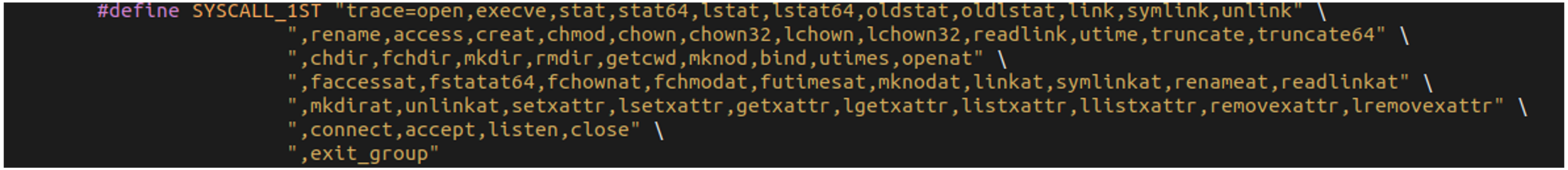
# Copy


PTU Package
Provenance DB
/data/ conf.dat
read()
/bin/ workflow
fork()
fork()
/data/ in.dat
/bin/ calculate
/bin/ reclass
read()
fork()
/bin/ filter
fork()
fork()
write()
/bin/ convert
/bin/ python
/data/ out.dat
fork()
$PKG_ROOT/
bin/
workflow
reclass
calculate
filter
convert
python
data/
conf.dat
in.dat
out.dat

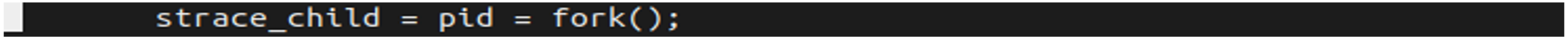
# Redirect with PID isolation



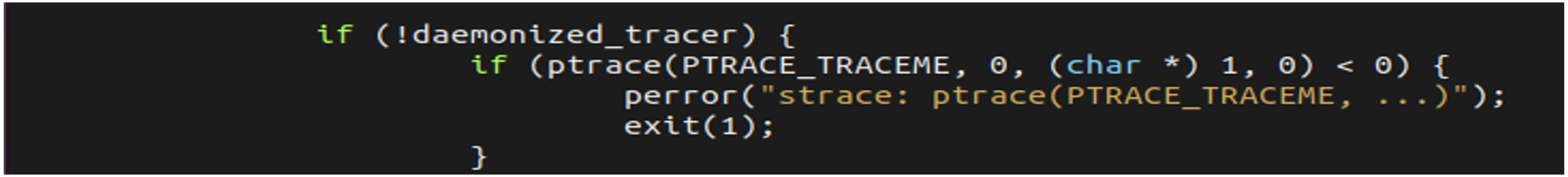
## Define syscalls to trace, fork child, and register trace



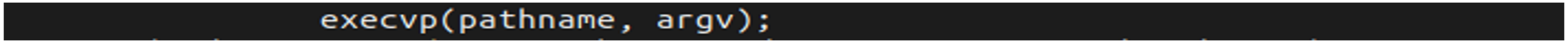
```
#define SYSCALL_1ST "trace=open,execve,stat,stat64,lstat,lstat64,oldstat,oldlstat,link,symlink,unlink" \
                ",rename,access,creat,chmod,chown,chown32,lchown,lchown32,readlink,utime,truncate,truncate64" \
                ",chdir,fchdir,mkdir,rmdir,getcwd,mknod,bind,utimes,openat" \
                ",faccessat,fstatat64,fchownat,fchmodat,futimesat,mknodat,linkat,symlinkat,renameat,readlinkat" \
                ",mkdirat,unlinkat,setxattr,lsetxattr,getxattr,lgetxattr,listxattr,llistxattr,removexattr,lremovexattr" \
                ",connect,accept,listen,close" \
                ",exit_group"
```



```
    strace_child = pid = fork();
```

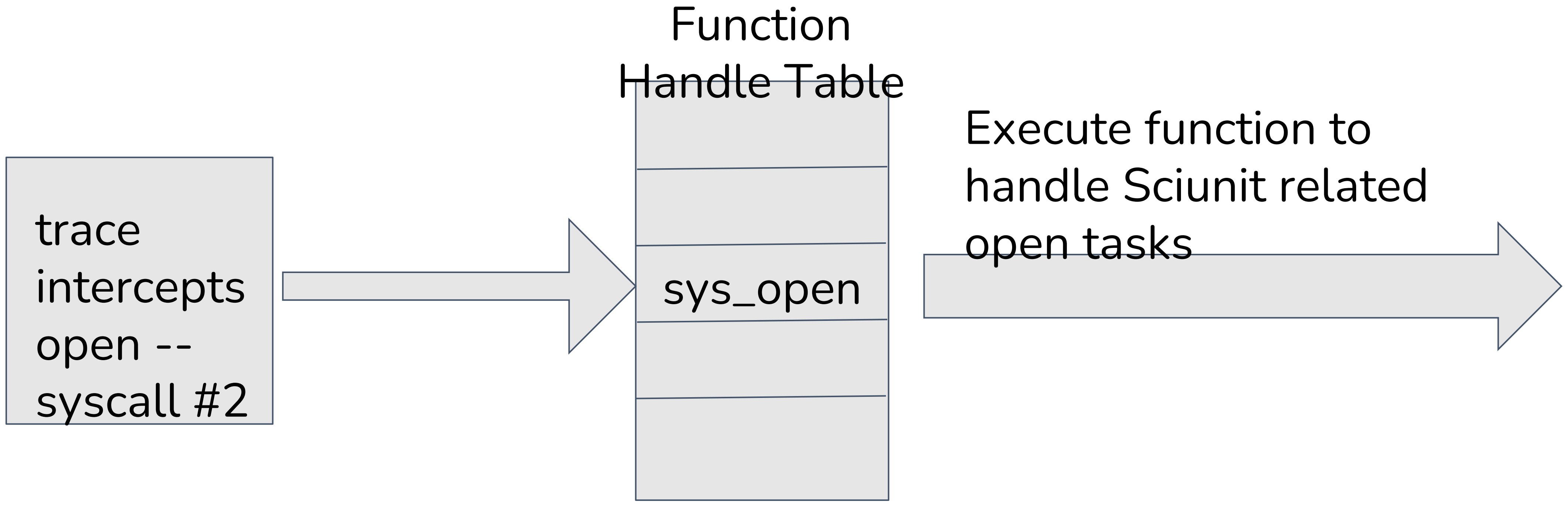


```
        if (!daemonized_tracer) {
                if (ptrace(PTRACE_TRACEME, 0, (char *) 1, 0) < 0) {
                        perror("strace: ptrace(PTRACE_TRACEME, ...)");
                        exit(1);
                }
```

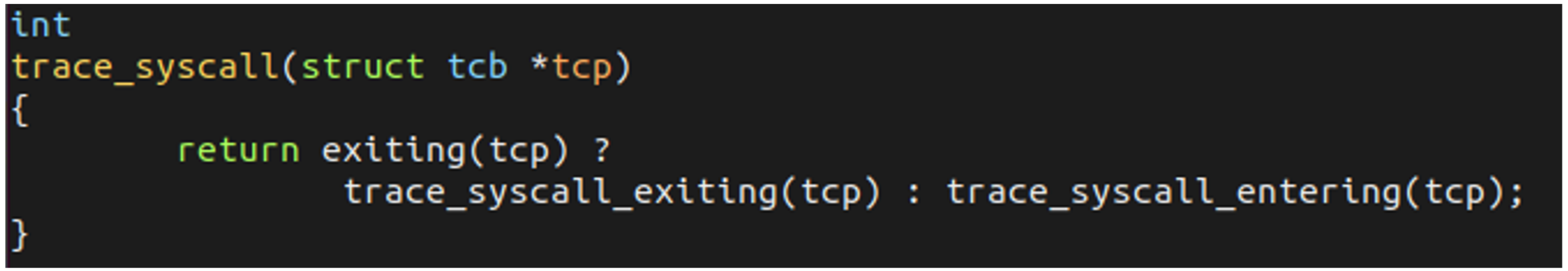


```
        execvp(pathname, argv);
```

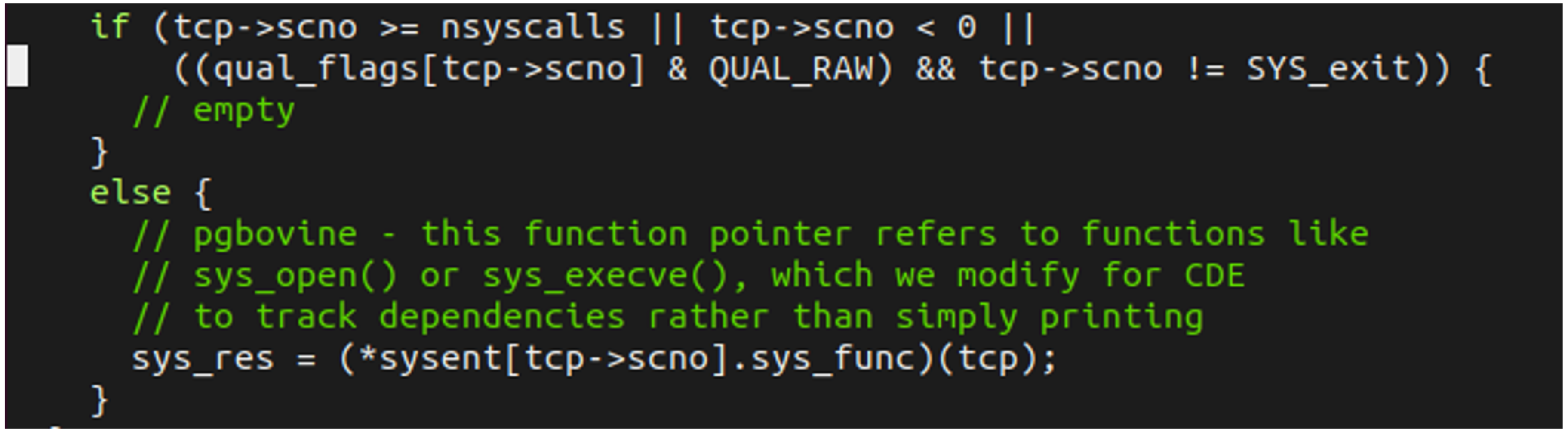
## Sciunit executes syscall specific functions dependent upon syscall


trace intercepts open -- syscall #2
Function Handle Table
sys_open
Execute function to handle Sciunit related open tasks

## Strace mods -- Use syscall # to look up function handle to execute

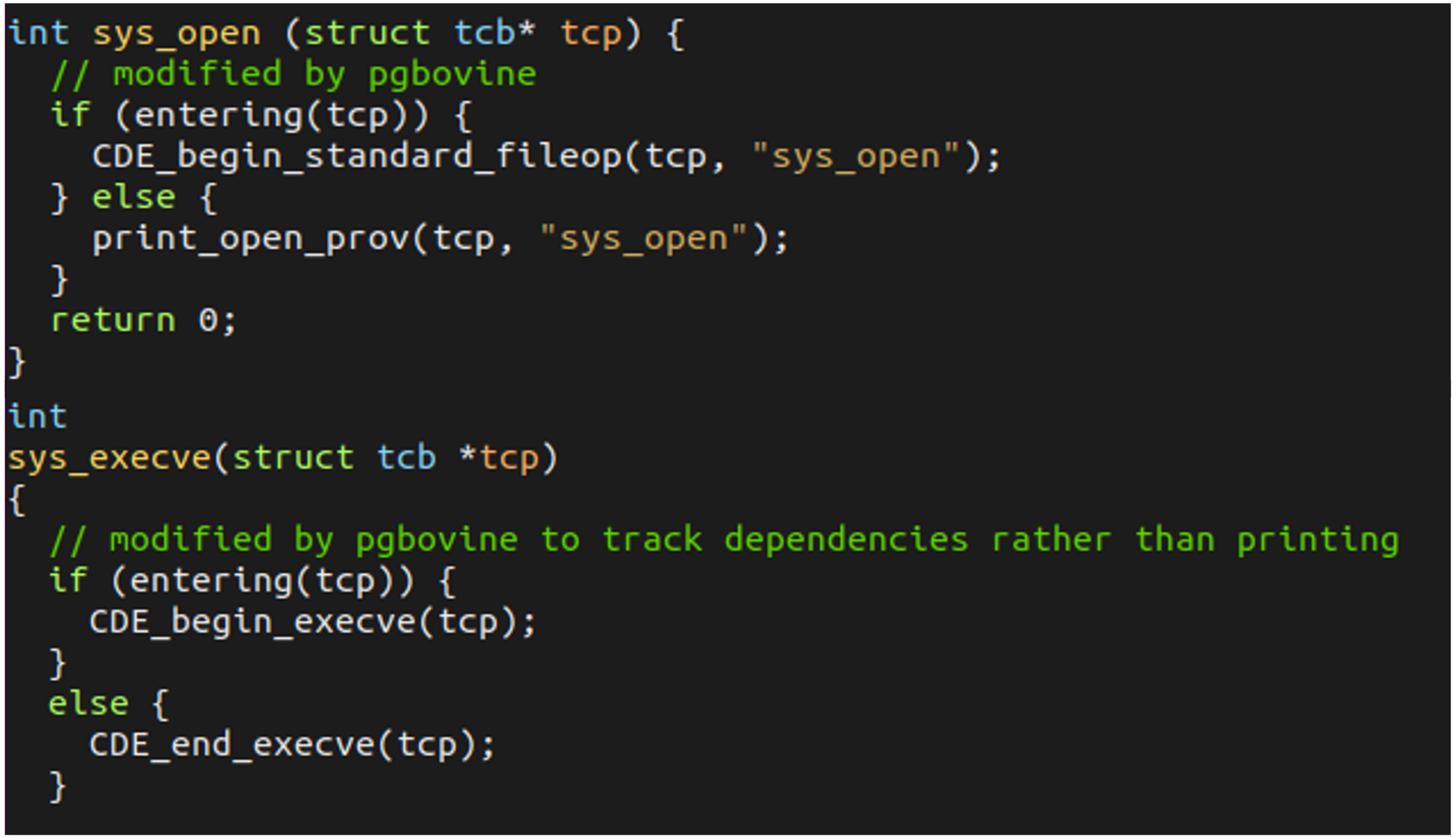


```
int
trace_syscall(struct tcb *tcp)
{
        return exiting(tcp) ?
                trace_syscall_exiting(tcp) : trace_syscall_entering(tcp);
}
```



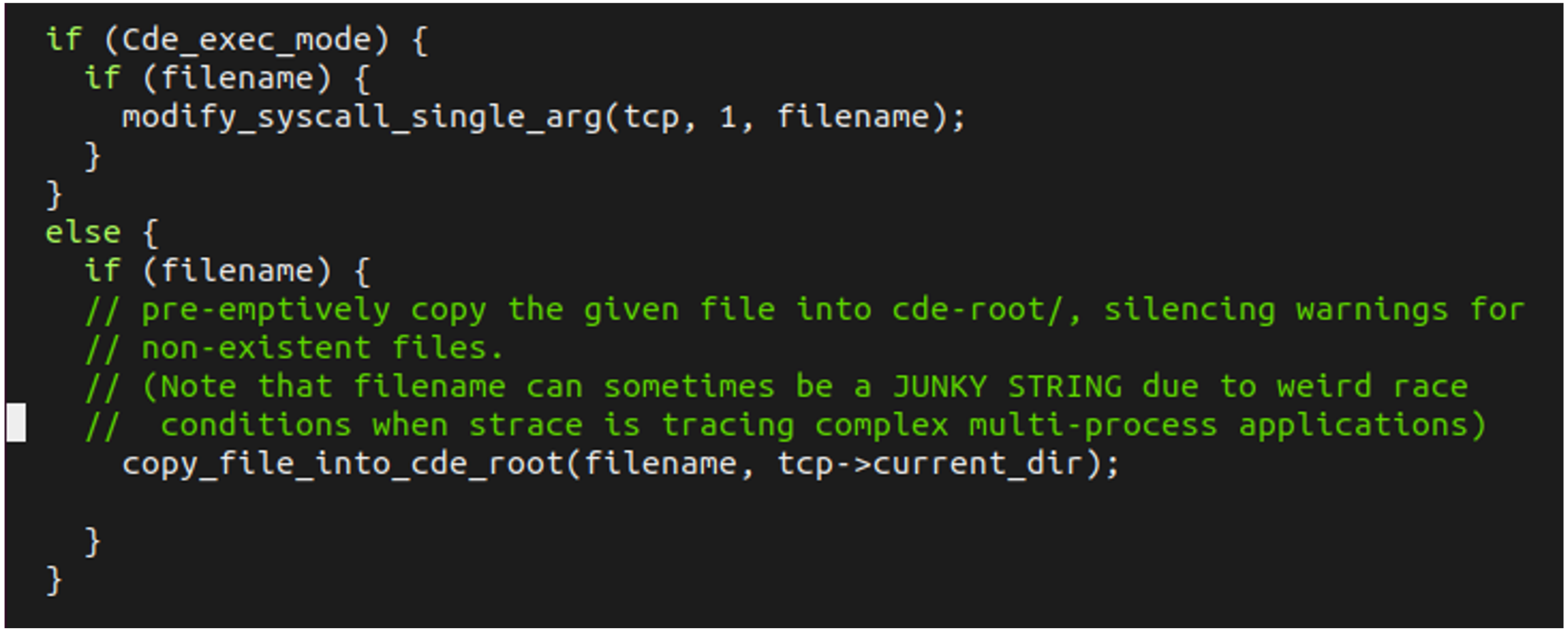
```
    if (tcp->scno >= nsyscalls || tcp->scno < 0 ||
        ((qual_flags[tcp->scno] & QUAL_RAW) && tcp->scno != SYS_exit)) {
      // empty
    }
    else {
      // pgbovine - this function pointer refers to functions like
      // sys_open() or sys_execve(), which we modify for CDE
      // to track dependencies rather than simply printing
      sys_res = (*sysent[tcp->scno].sys_func)(tcp);
    }
```

## Each traced syscall has its own handler function



```
int sys_open (struct tcb* tcp) {
  // modified by pgbovine
  if (entering(tcp)) {
    CDE_begin_standard_fileop(tcp, "sys_open");
  } else {
    print_open_prov(tcp, "sys_open");
  }
  return 0;
}
int
sys_execve(struct tcb *tcp)
{
  // modified by pgbovine to track dependencies rather than printing
  if (entering(tcp)) {
    CDE_begin_execve(tcp);
  }
  else {
    CDE_end_execve(tcp);
  }
```

## sys_open – begin_standard_fileop()



```
if (Cde_exec_mode) {
  if (filename) {
    modify_syscall_single_arg(tcp, 1, filename);
  }
}
else {
  if (filename) {
  // pre-emptively copy the given file into cde-root/, silencing warnings for
  // non-existent files.
  // (Note that filename can sometimes be a JUNKY STRING due to weird race
  //  conditions when strace is tracing complex multi-process applications)
    copy_file_into_cde_root(filename, tcp->current_dir);

  }
}
```

## Sciunit

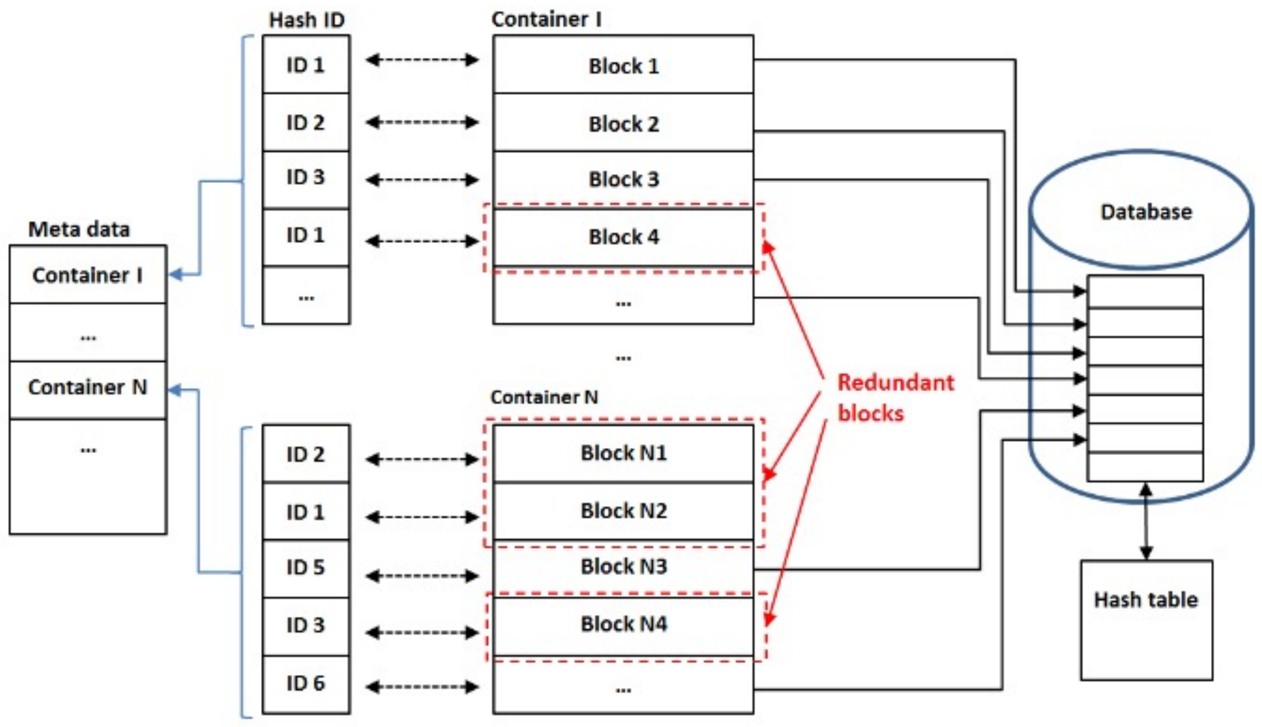
- *sciunit-exec*
  - Build a package of authors’ source code, data, and environment variables
  - Record process- and file-level details about a reference execution
- *sciunit-repeat*
  - In own PID namespace
  - Re-execute specified part of the provenance graph

## Sciunit containers

- All needed files (including binaries) are stored in the container during capture
- During repeat the containerized files and binaries are referenced
- Containers can be distributed to others
- The exact versions of each file referenced in the original are maintained regardless of changes in environment.

## Storage

- content-based deduplication



## Sciunit uses

- Make it easier for user to preserve a computational experiment
- Make it easier for user to share a computational environment
- Make it easier for user to repeat a published experiment
- Make it easier for user to extend or modify a published experiment
- Make it easier for user to understand how a result is produced

## Sciunit

- https://sciunit.run/
- https://github.com/depaul-dice/sciunit