Hyperkernel: Push-Button Verification of an OS Kernel

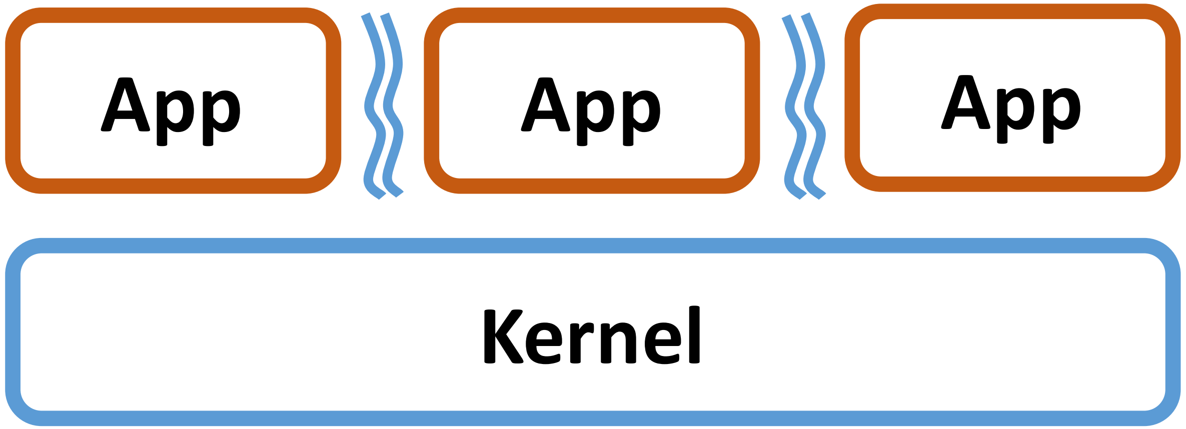
Luke Nelson, **Helgi Sigurbjarnarson**, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang

UNIVERSITY *of* WASHINGTON

PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

The OS Kernel is a critical component

- Essential for application correctness and security
- Kernel bugs can compromise the entire system



:(

Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (0% complete)

If you'd like to know more, you can search online later for this error: HAL_INITIALIZATION_FAILED

:(

Your P  e just
collect u. (0%
compl

If you'd like to

Formal verification: high correctness assurance

- Write a spec of expected behavior
- Prove that implementation matches the spec

seL4

IronClad

CERTIKOS

- **Goal: How much can we minimize the proof burden**

Formal verification: high correctness assurance

- Write a spec of expected behavior
- Prove that implementation matches the spec

Proof effort:
11 person years

IronClad CertiKOS

- **Goal: How much can we minimize the proof burden**

Our result: Hyperkernel

- **Unix-like OS kernel:** based on xv6
- **Fully automated verification using the Z3 solver**
  - Functional correctness of system calls
  - Crosscutting properties (e.g., process isolation)
- **Limitations:**
  - Uniprocessor
  - Initialization & glue code unverified

Designing Hyperkernel for proof automation

Xv6

Hyperkernel

- Syscall semantics are loop-y and require writing loop invariants
- Kernel pointers difficult to reason about
- C is difficult to model

Designing Hyperkernel for proof automation

Xv6

- Syscall semantics are loop-y and require writing loop invariants
- Kernel pointers difficult to reason about
- C is difficult to model

Hyperkernel

- Finite interface

Designing Hyperkernel for proof automation

Xv6

- Syscall semantics are loop-y and require writing loop invariants
- Kernel pointers difficult to reason about
- C is difficult to model

Hyperkernel

- Finite interface
- Separate user/kernel spaces and use identity mapping for kernel

Designing Hyperkernel for proof automation

Xv6

- Syscall semantics are loop-y and require writing loop invariants
- Kernel pointers difficult to reason about
- C is difficult to model

Hyperkernel

- Finite interface
- Separate user/kernel spaces and use identity mapping for kernel
- Verify LLVM intermediate representation (IR)

Designing Hyperkernel for proof automation

Xv6	**Hyperkernel**
• Syscall semantics are loop-y and require writing loop invariants	• Finite interface
• Kernel pointers difficult to reason about	• Separate user/kernel spaces and use identity mapping for kernel
• C is difficult to model	• Verify LLVM intermediate representation (IR)

Outline

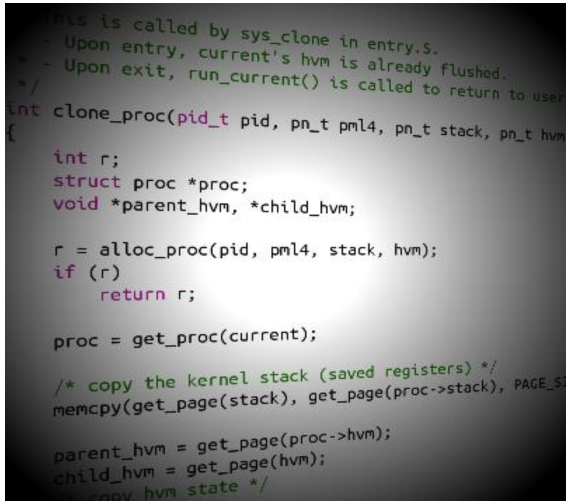
- Verification workflow
- Finite interface design
- Demo
- Evaluation & lessons learned

Outline

- **Verification workflow**
- Finite interface design
- Demo
- Evaluation & lessons learned

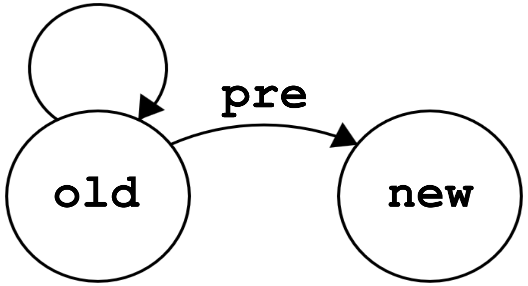
Overview of verification workflow

Syscall Implementation

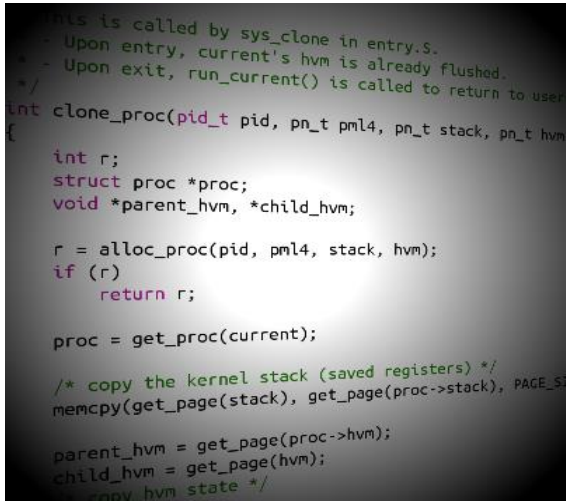


Overview of verification workflow

State Machine Specification

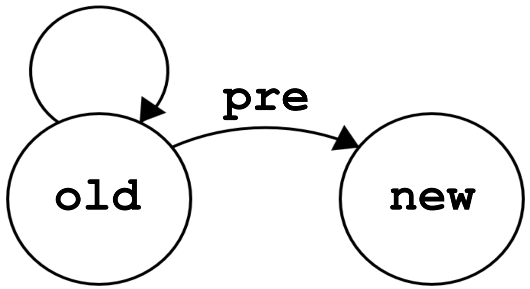


Syscall Implementation

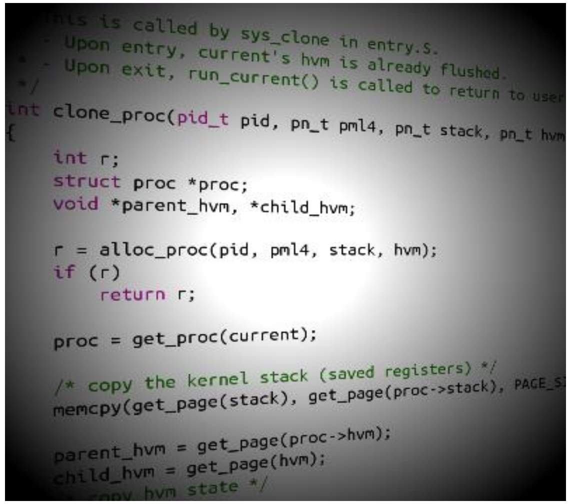


Overview of verification workflow

State Machine Specification



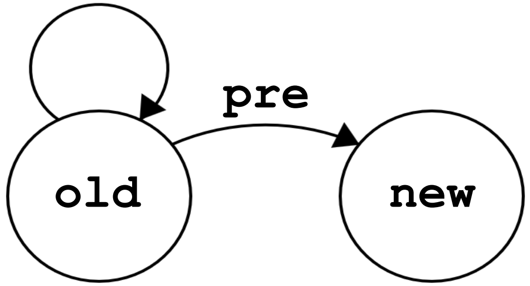
Syscall Implementation



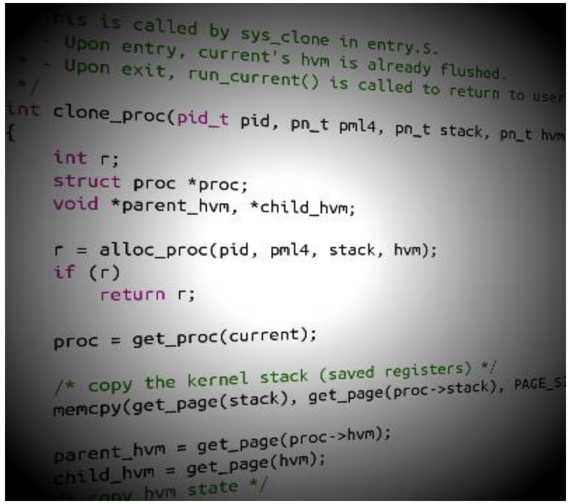

```
def sys_set_runnable(old, pid):
    pre = old.procs[pid].state == PROC_EMBRYO
    new = old.copy()
    new.procs[pid].state = PROC_RUNNABLE
    return pre, new
```

Overview of verification workflow

State Machine Specification



Syscall Implementation

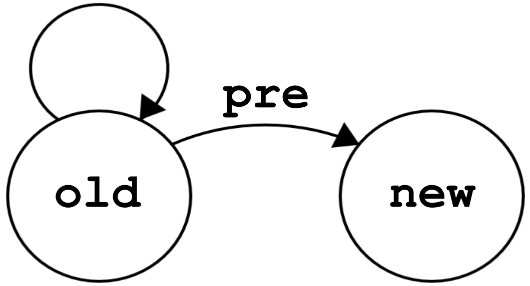


```
 * This is called by sys_clone in entry.S.
 * - Upon entry, current's hvm is already flushed.
 * - Upon exit, run_current() is called to return to user
 */
int clone_proc(pid_t pid, pn_t pml4, pn_t stack, pn_t hvm
{
    int r;
    struct proc *proc;
    void *parent_hvm, *child_hvm;

    r = alloc_proc(pid, pml4, stack, hvm);
    if (r)
        return r;

    proc = get_proc(current);

    /* copy the kernel stack (saved registers) */
    memcpy(get_page(stack), get_page(proc->stack), PAGE_S

    parent_hvm = get_page(proc->hvm);
    child_hvm = get_page(hvm);
    /* copy hvm state */
```

```python
def sys_set_runnable(old, pid):
    pre = old.procs[pid].state == PROC_EMBRYO
    new = old.copy()
    new.procs[pid].state = PROC_RUNNABLE
    return pre, new
```

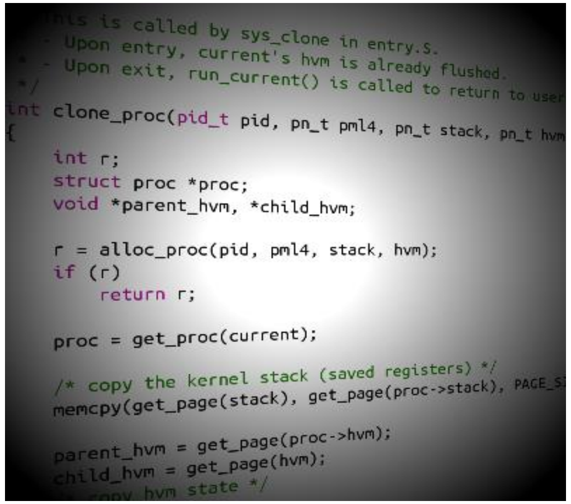
Overview of verification workflow

State Machine Specification




```python
def sys_set_runnable(old, pid):
    pre = old.procs[pid].state == PROC_EMBRYO
    new = old.copy()
    new.procs[pid].state = PROC_RUNNABLE
    return pre, new
```

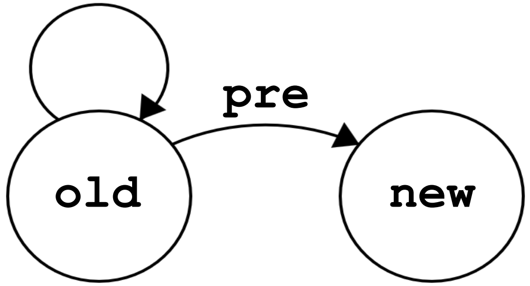
Syscall Implementation



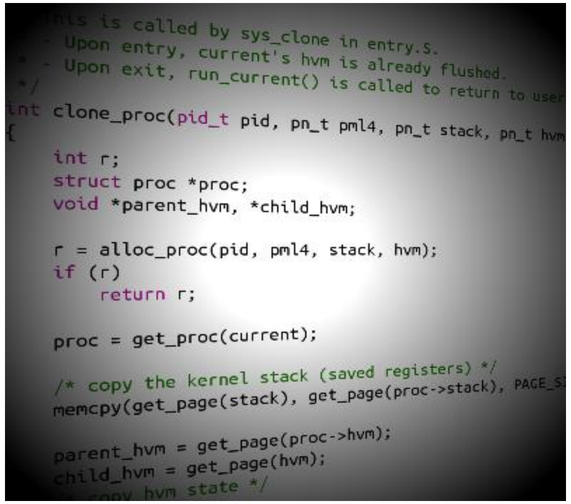
```c
 * This is called by sys_clone in entry.S.
 * - Upon entry, current's hvm is already flushed.
 * - Upon exit, run_current() is called to return to user
 */
int clone_proc(pid_t pid, pn_t pml4, pn_t stack, pn_t hvm
{
    int r;
    struct proc *proc;
    void *parent_hvm, *child_hvm;

    r = alloc_proc(pid, pml4, stack, hvm);
    if (r)
        return r;

    proc = get_proc(current);

    /* copy the kernel stack (saved registers) */
    memcpy(get_page(stack), get_page(proc->stack), PAGE_S

    parent_hvm = get_page(proc->hvm);
    child_hvm = get_page(hvm);
    /* copy hvm state */
```

Overview of verification workflow

State Machine Specification



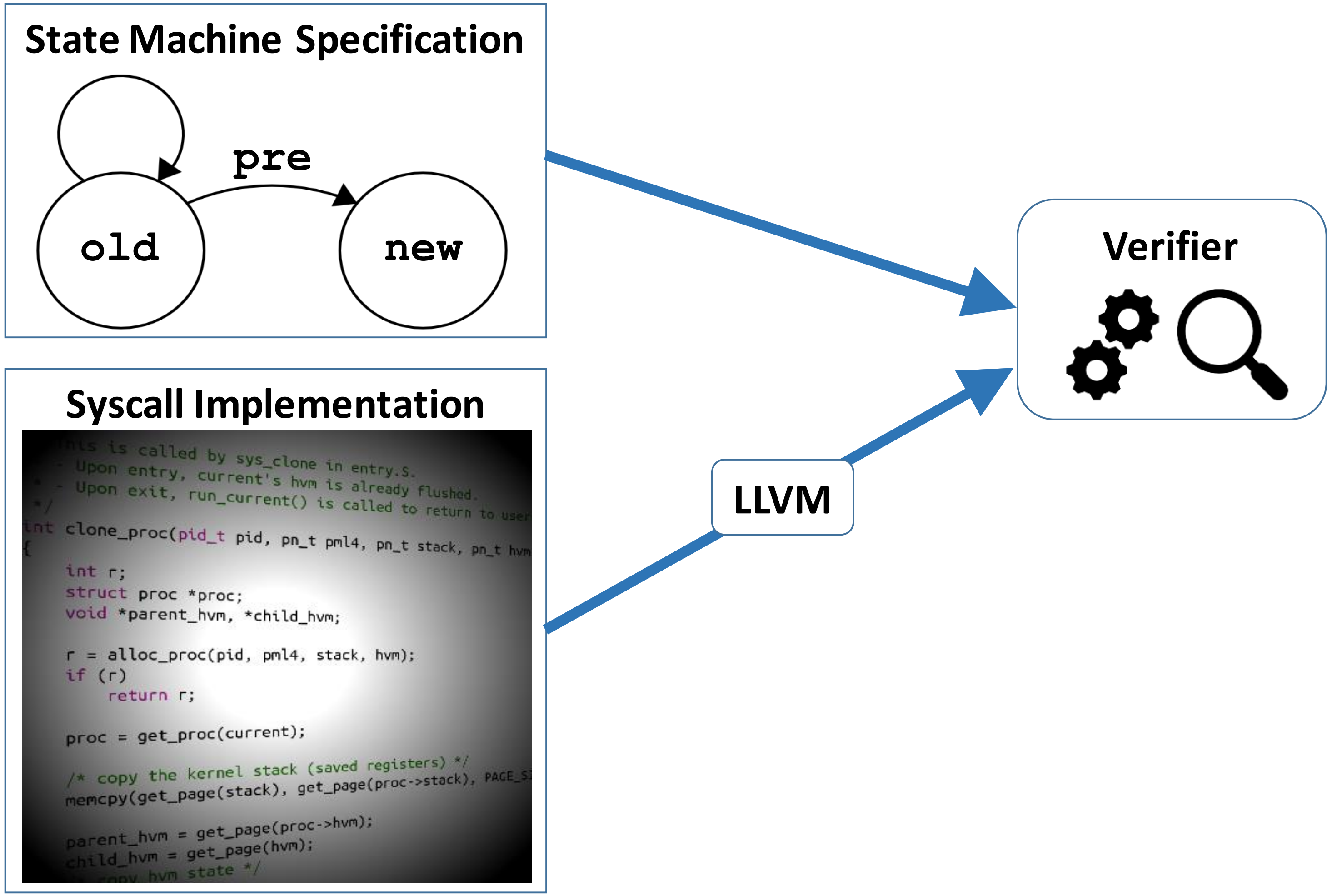
Syscall Implementation



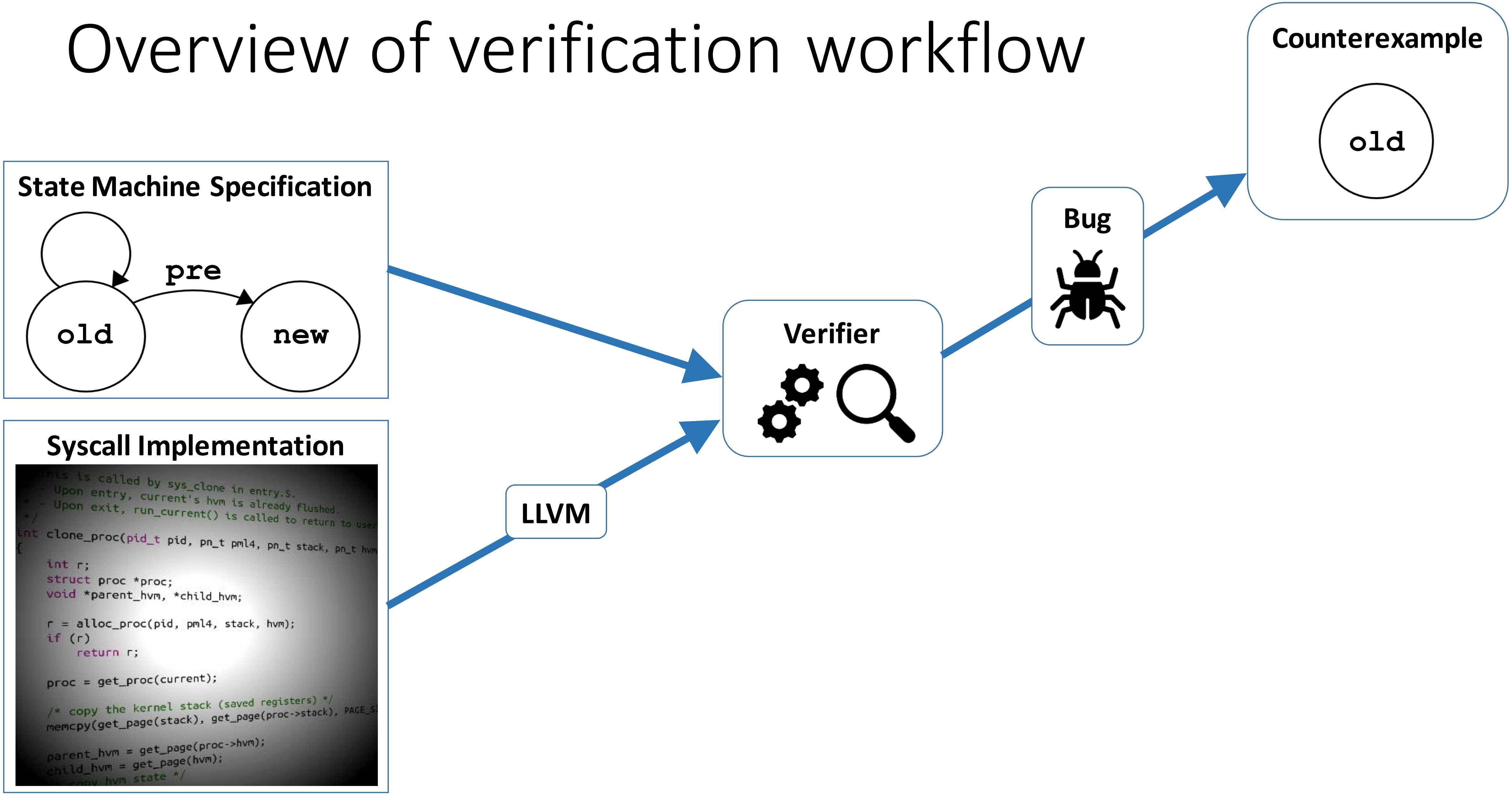
```
int clone_proc(pid_t pid, pn_t pml4, pn_t stack, pn_t hvm
{
    int r;
    struct proc *proc;
    void *parent_hvm, *child_hvm;

    r = alloc_proc(pid, pml4, stack, hvm);
    if (r)
        return r;

    proc = get_proc(current);

    /* copy the kernel stack (saved registers) */
    memcpy(get_page(stack), get_page(proc->stack), PAGE_S

    parent_hvm = get_page(proc->hvm);
    child_hvm = get_page(hvm);
```

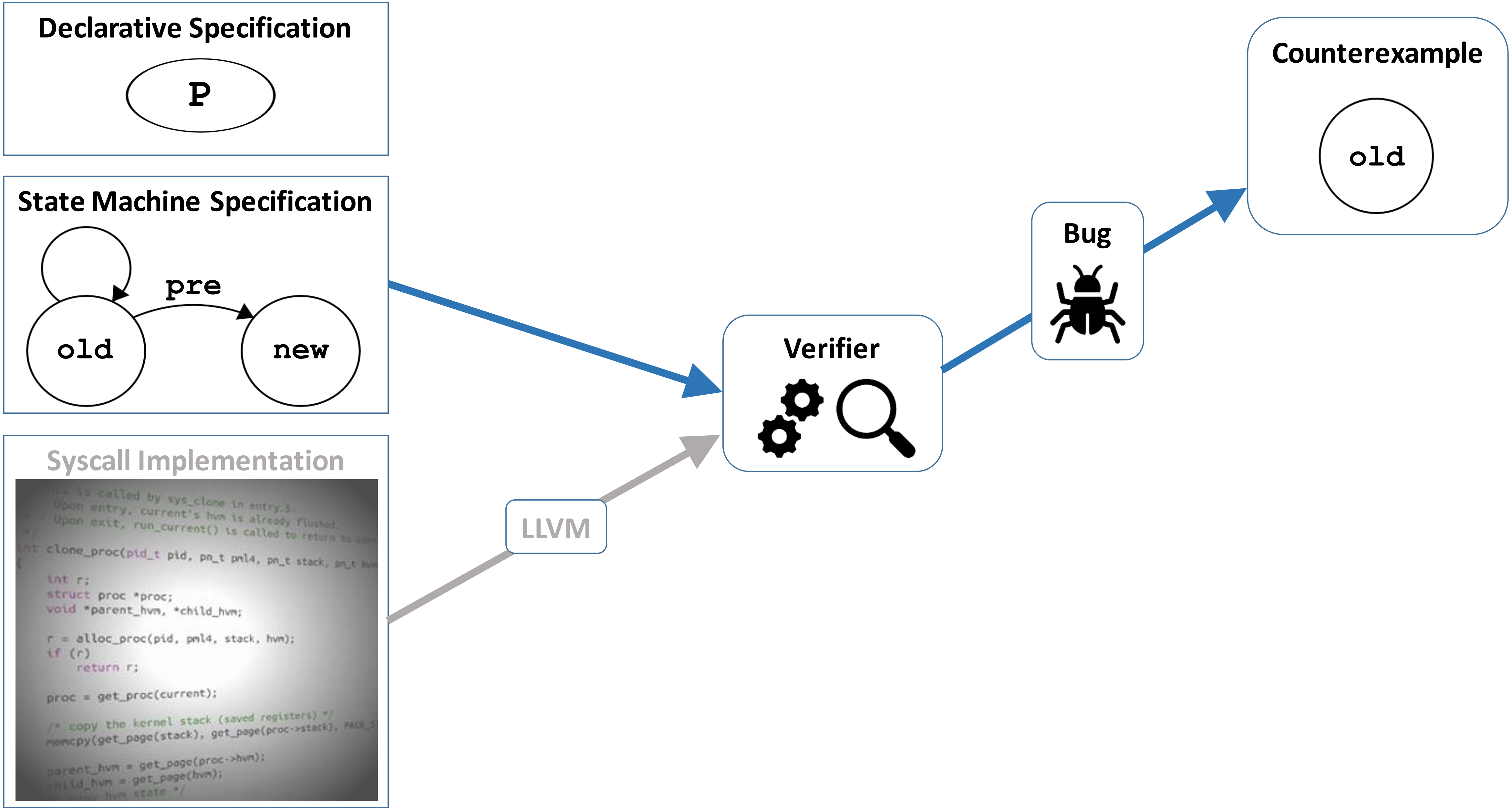
```
def sys_set_runnable(old, pid):
    pre = old.procs[pid].state == PROC_EMBRYO
    new = old.copy()
    new.procs[pid].state = PROC_RUNNABLE
    return pre, new
```

Overview of verification workflow

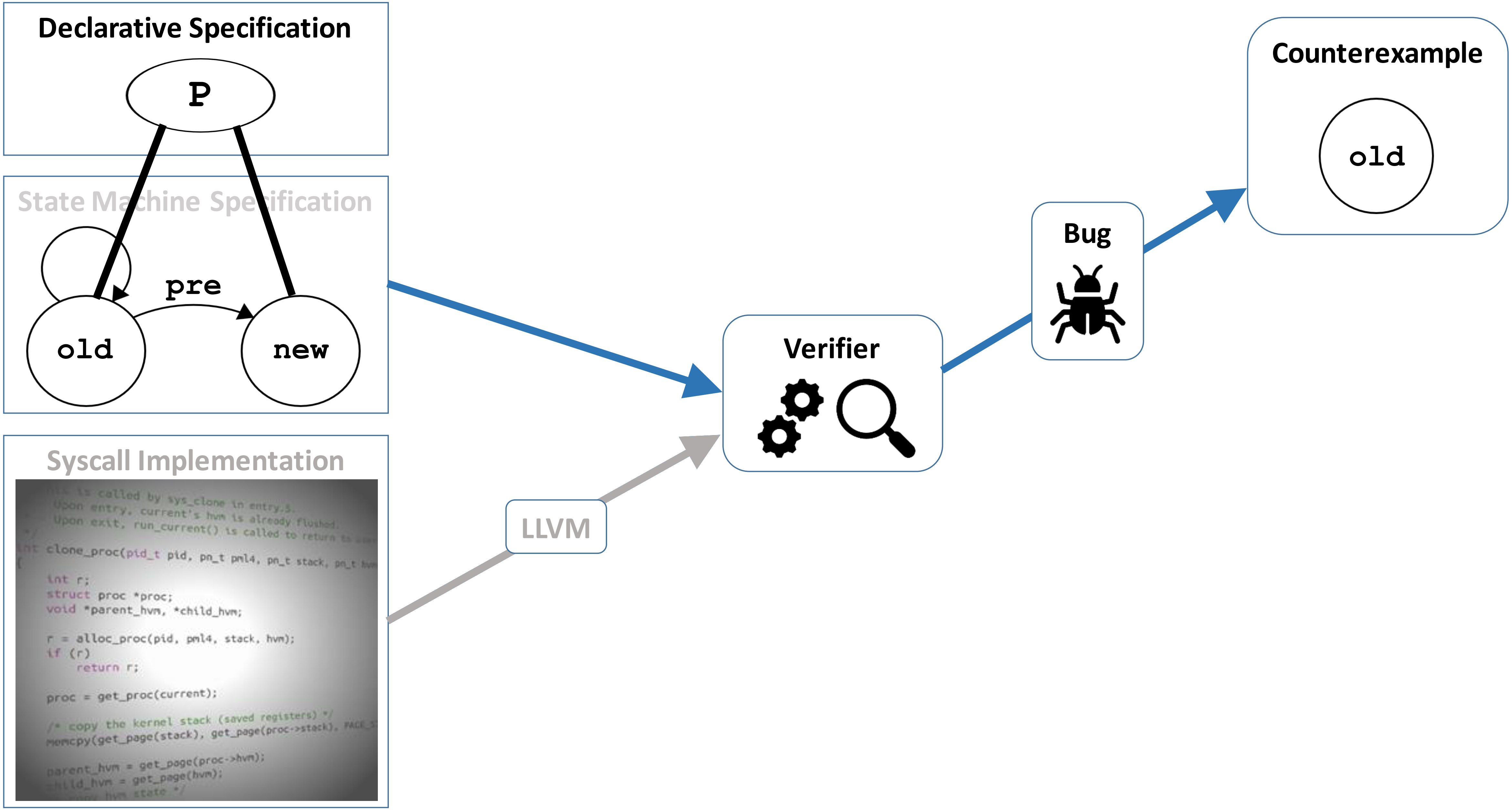


Overview of verification workflow





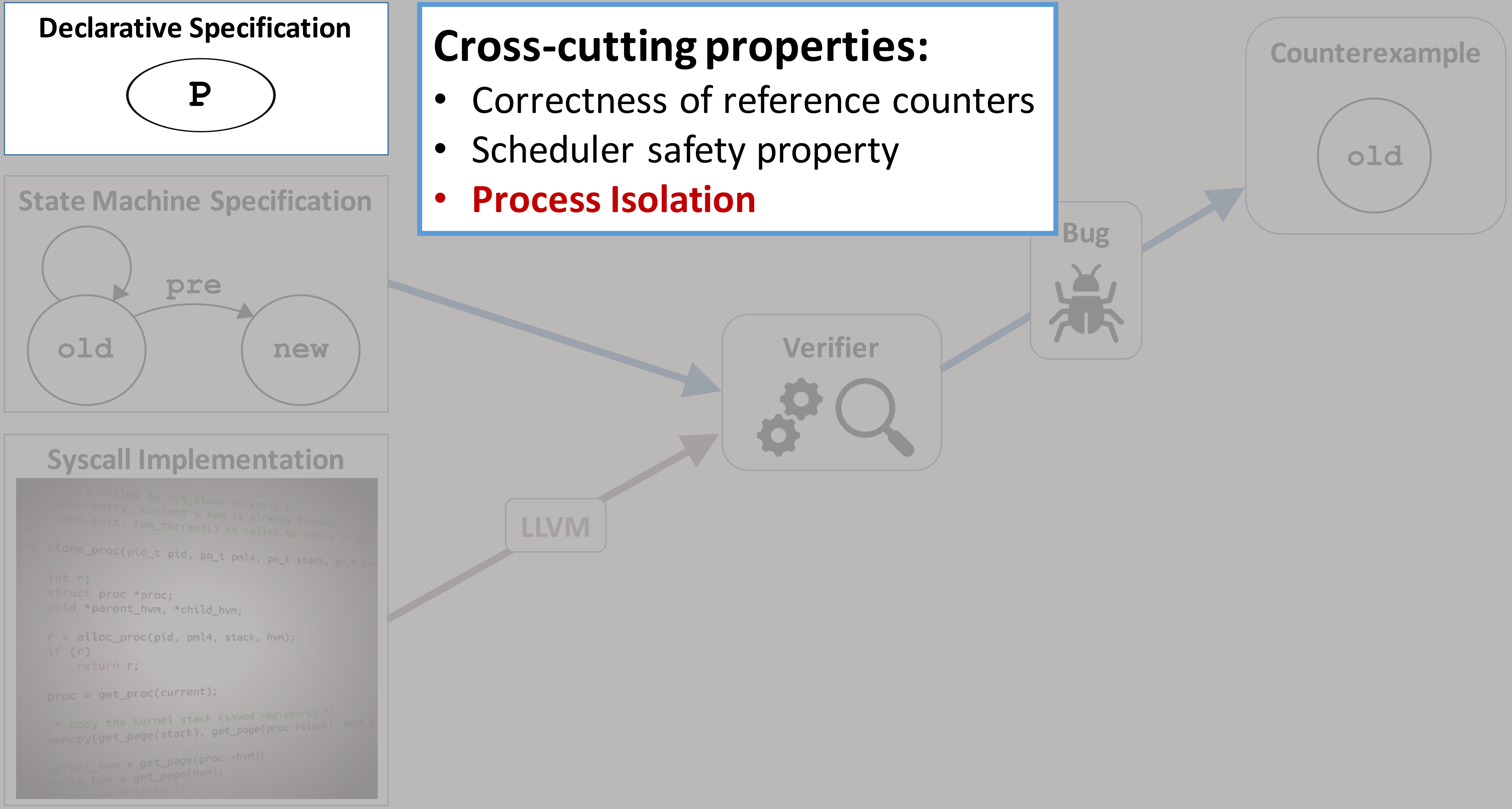
Declarative Specification
P
State Machine Specification
pre
old
new
Syscall Implementation
int clone_proc(pid_t pid, pn_t pml4, pn_t stack,
int r;
struct proc *proc;
void *parent_hvm, *child_hvm;
r = alloc_proc(pid, pml4, stack, hvm);
if (r)
return r;
proc = get_proc(current);
/* copy the kernel stack (saved registers) */
memcpy(get_page(stack), get_page(proc->stack),
parent_hvm = get_page(proc->hvm);
child_hvm = get_page(hvm);
LLVM
Verifier
Bug
Counterexample
old

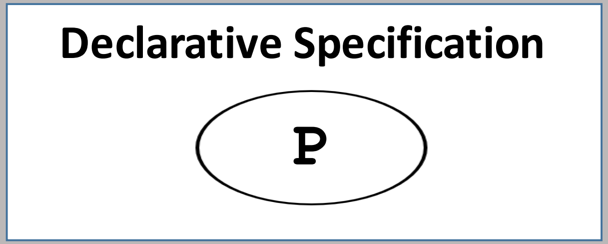


Declarative Specification
P
State Machine Specification
pre
old
new
Syscall Implementation
LLVM
Verifier
Bug
Counterexample
old

Cross-cutting properties:

- Correctness of reference counters
- Scheduler safety property
- Process Isolation

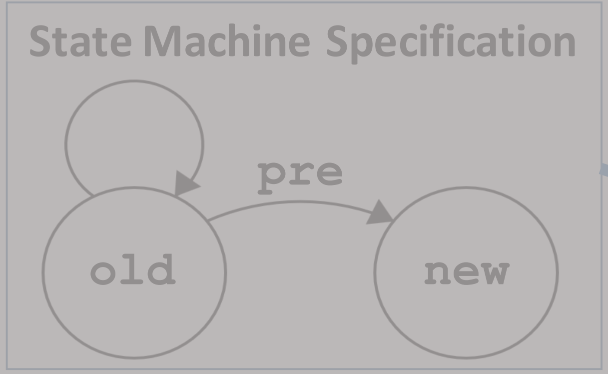


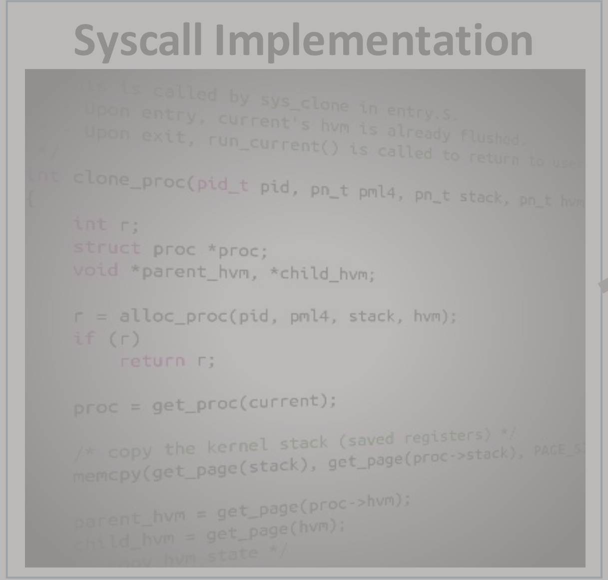


Cross-cutting properties:

- Correctness of reference counters
- Scheduler safety property
- **Process Isolation**

For any virtual address in a process p,
if the virtual address maps to a page
the page must be exclusively owned by p.





LLVM

Bug

Counterexample

old

Declarative Specification

P

State Machine Specification

pre

old

new

Syscall Implementation

Bug

Counterexample

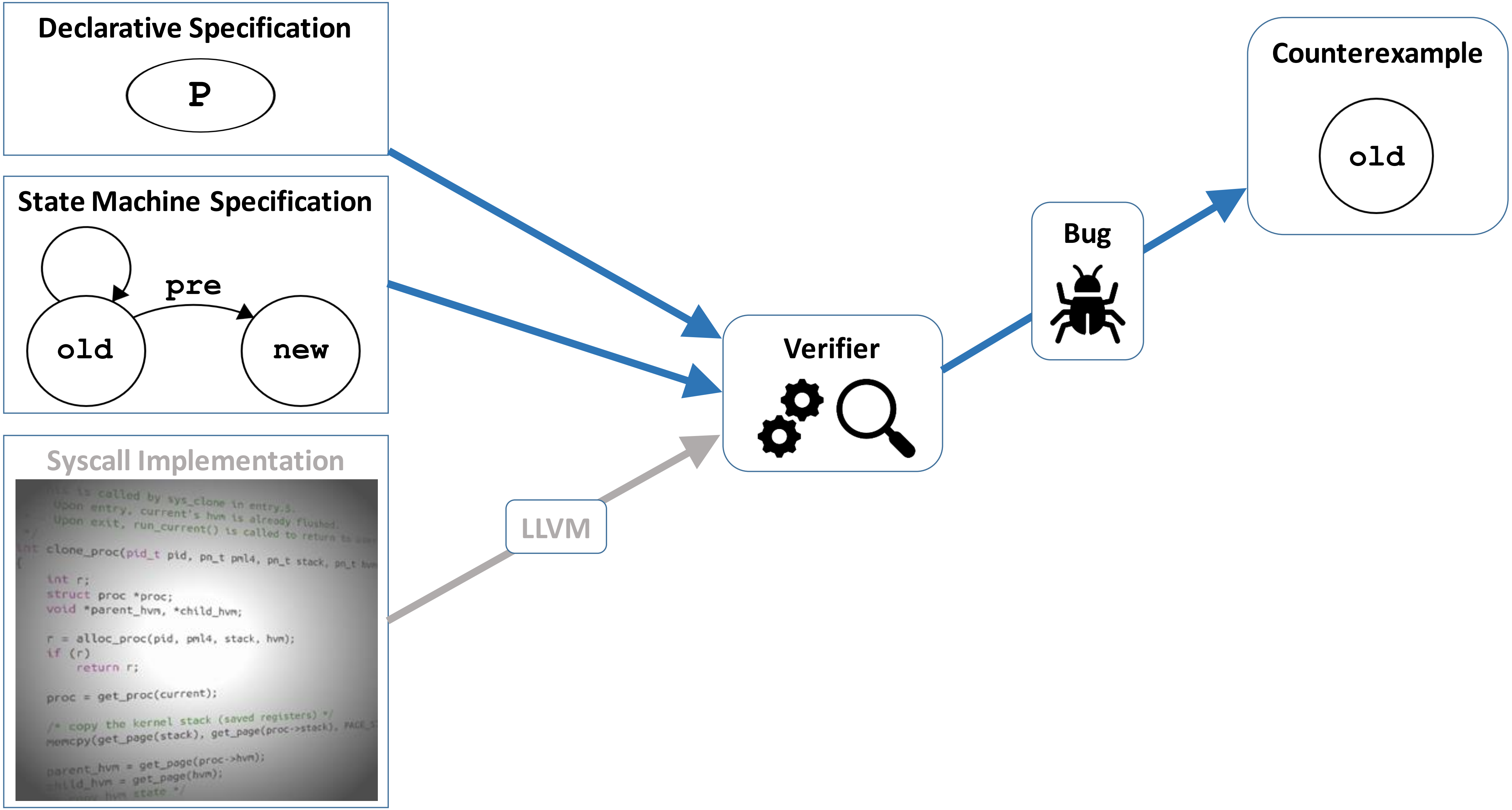
old

Cross-cutting properties:

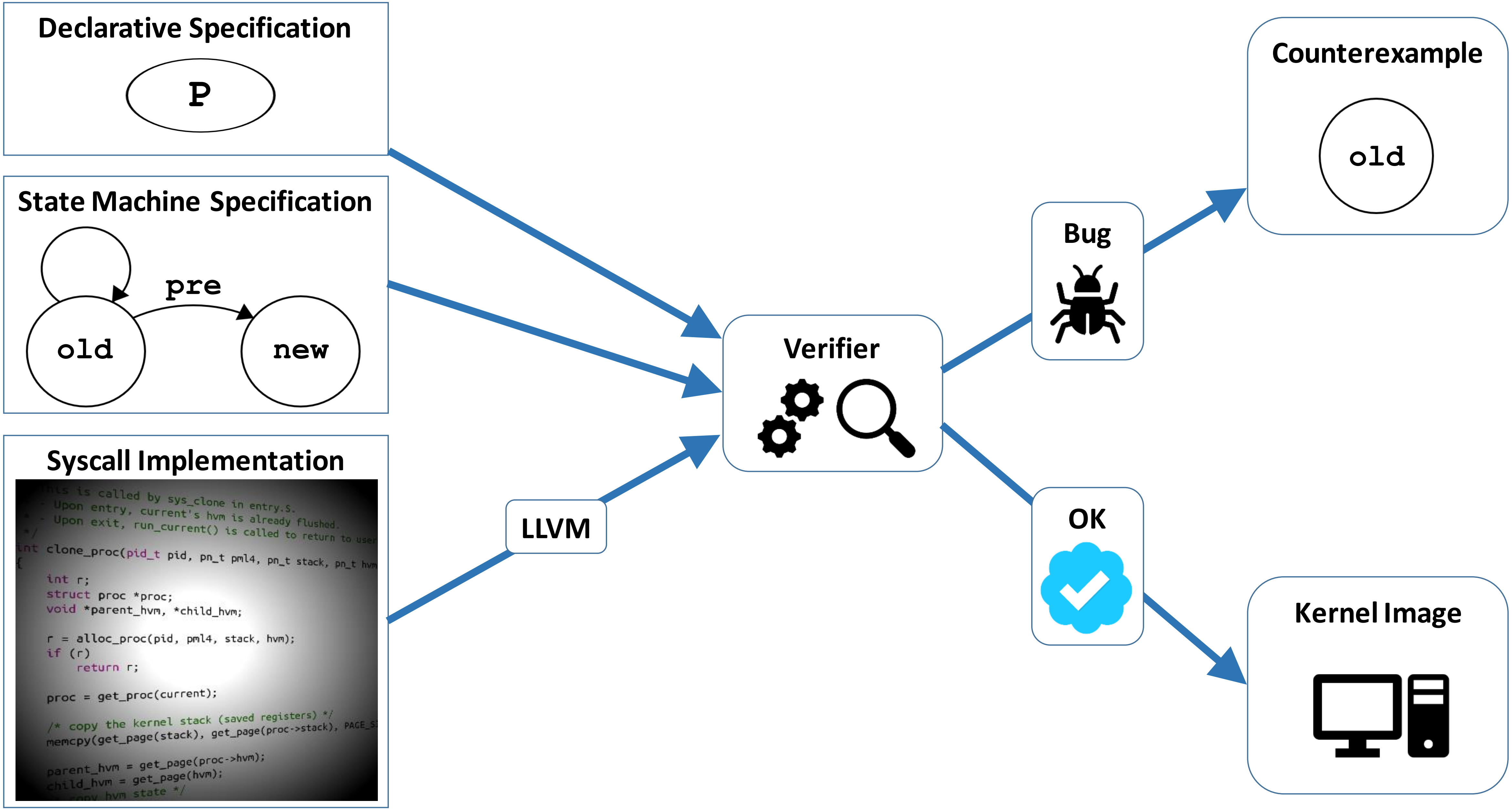
- Correctness of reference counters
- Scheduler safety property
- **Process Isolation**

For any virtual address in a process p,
if the virtual address maps to a page
the page must be exclusively owned by p.

```
page, success = page_walk(state, pid, va)
isolation = Implies(success,
                    state.pages[page].owner == pid)

Show: ForAll([pid, va], isolation)
```



Declarative Specification
P
State Machine Specification
pre
old
new
Syscall Implementation
int clone_proc(pid_t pid, pn_t pml4, pn_t stack,
int r;
struct proc *proc;
void *parent_hvm, *child_hvm;
r = alloc_proc(pid, pml4, stack, hvm);
if (r)
return r;
proc = get_proc(current);
/* copy the kernel stack (saved registers) */
memcpy(get_page(stack), get_page(proc->stack),
parent_hvm = get_page(proc->hvm);
child_hvm = get_page(hvm);
LLVM
Verifier
Bug
Counterexample
old


Declarative Specification
P
State Machine Specification
pre
old
new
Syscall Implementation
int clone_proc(pid_t pid, pn_t pml4, pn_t stack,
int r;
struct proc *proc;
void *parent_hvm, *child_hvm;
r = alloc_proc(pid, pml4, stack, hvm);
if (r)
return r;
proc = get_proc(current);
/* copy the kernel stack (saved registers) */
memcpy(get_page(stack), get_page(proc->stack),
parent_hvm = get_page(proc->hvm);
child_hvm = get_page(hvm);
LLVM
Verifier
Bug
OK
Counterexample
old
Kernel Image

Outline

- Verification workflow
- **Finite interface design**
- Demo
- Evaluation & lessons learned

Verification through symbolic execution

- **Goal:** Minimize proof burden
  - No manual proofs or code annotations

- **Symbolic execution**
  - Fully automated technique, used in bug-finding
  - Full functional verification if program is free of loops and state is finite
  - Feasible when units of work sufficiently small for solving

- **Hyperkernel approach:** Finite interface design

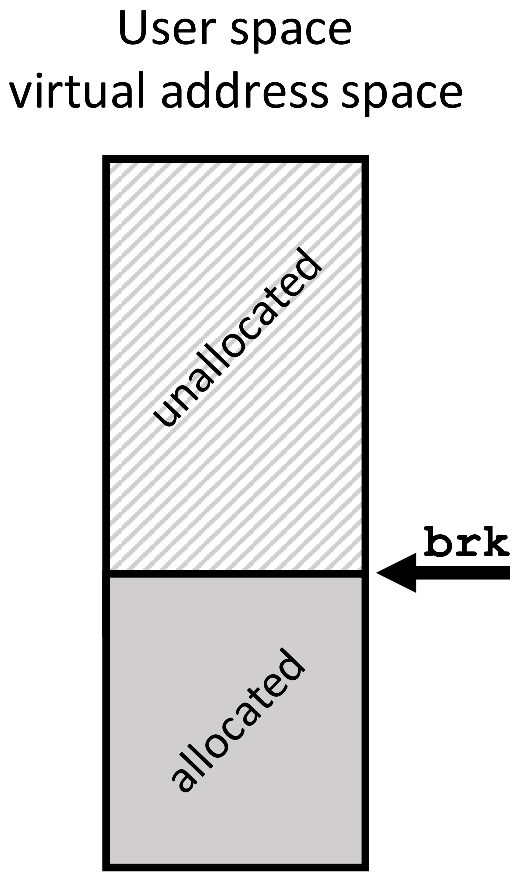
Overview of techniques

- Safely push loops into user space
- Explicit resource management
- Decompose complex syscalls
- Validate linked data structures
- Smart SMT encodings

Overview of techniques

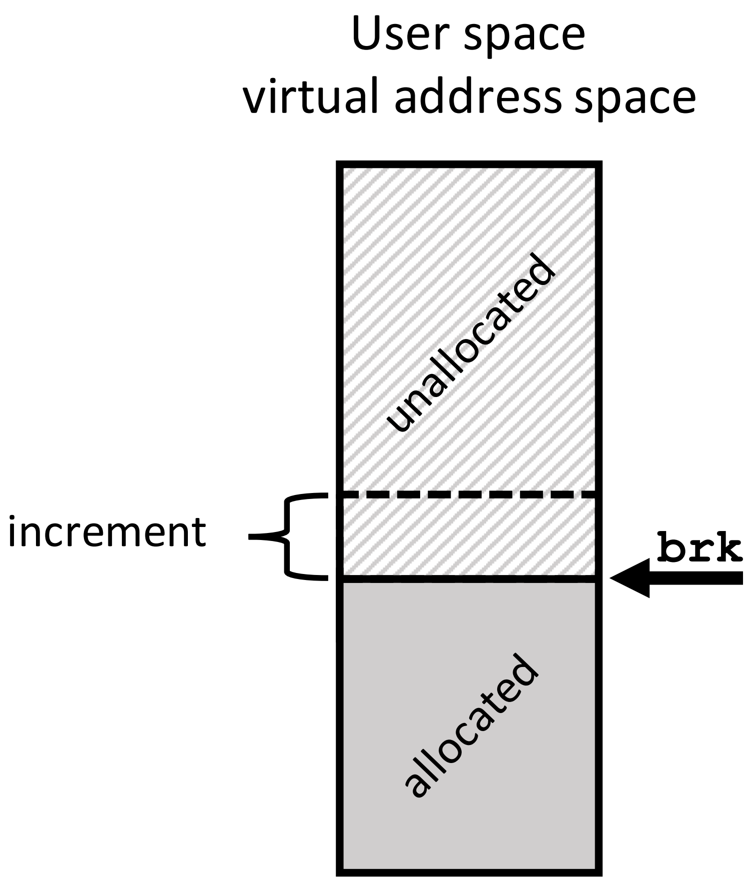
- **Safely push loops into user space**
- **Explicit resource management**
- **Decompose complex syscalls**
- Validate linked data structures
- Smart SMT encodings

The sbrk() system call




```
void *sbrk(intptr_t increment)
```

The sbrk() system call

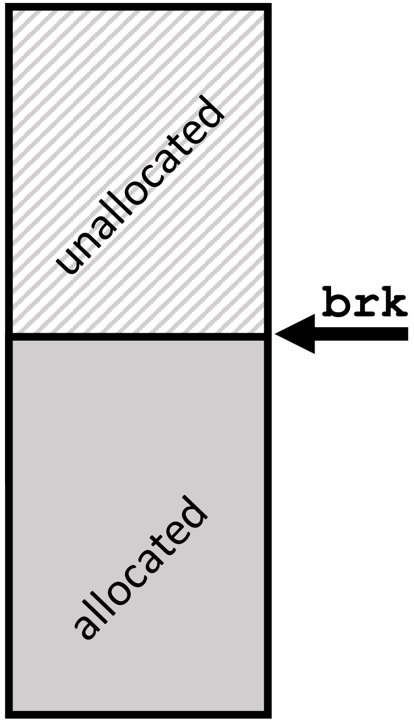



```
void *sbrk(intptr_t increment)
```

increments the programs data space by increment bytes

The sbrk() system call

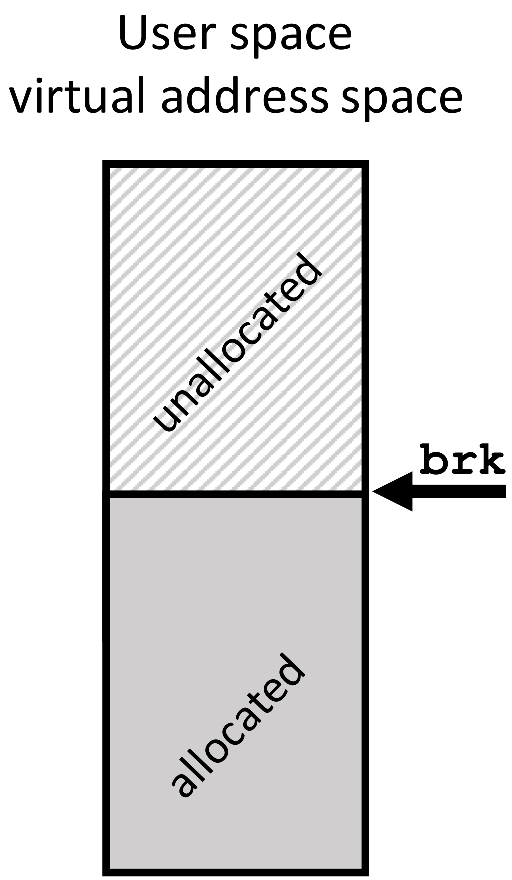
User space
virtual address space




```
void *sbrk(intptr_t increment)
```

increments the programs data space by increment bytes

The sbrk() system call




```
void *sbrk(intptr_t increment)
```

increments the programs data space by increment bytes

Goal: Redesign sbrk(); ensuring process isolation.

The sbrk() system call: Dealing with loops

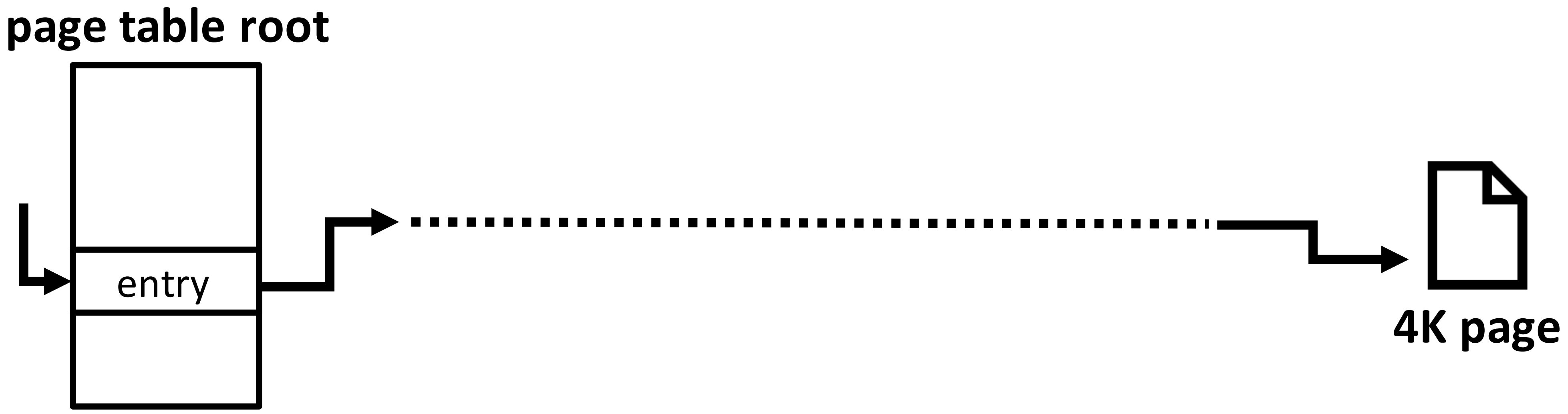
```
void *sbrk(intptr_t increment)
```

The sbrk() system call: Dealing with loops

```
void *sbrk(intptr_t increment)
```

The sbrk() system call: Dealing with loops

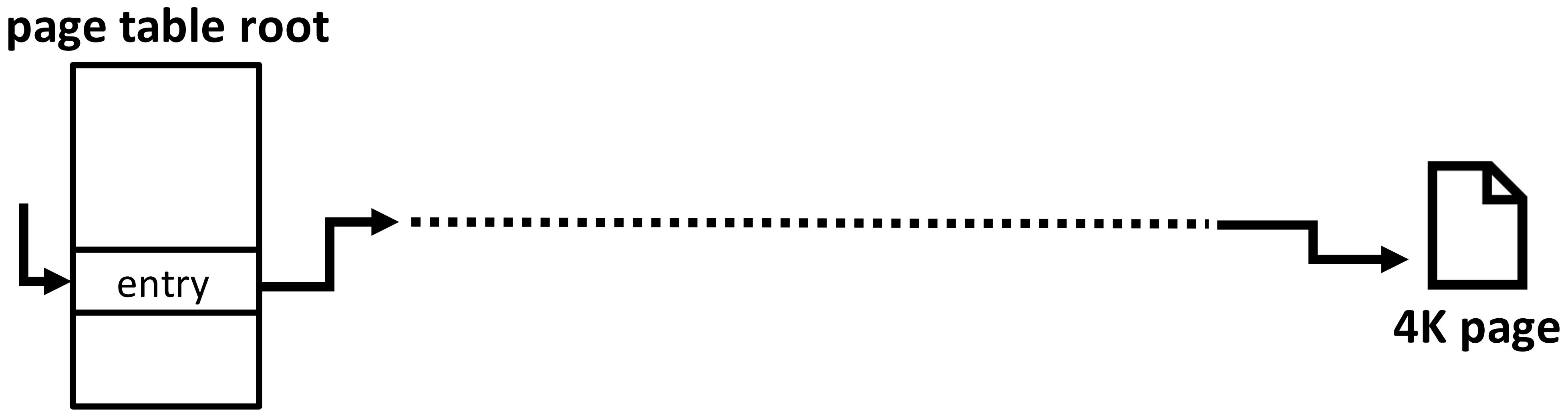
```
void *sbrk(intptr_t increment)
```



The sbrk() system call: Dealing with loops

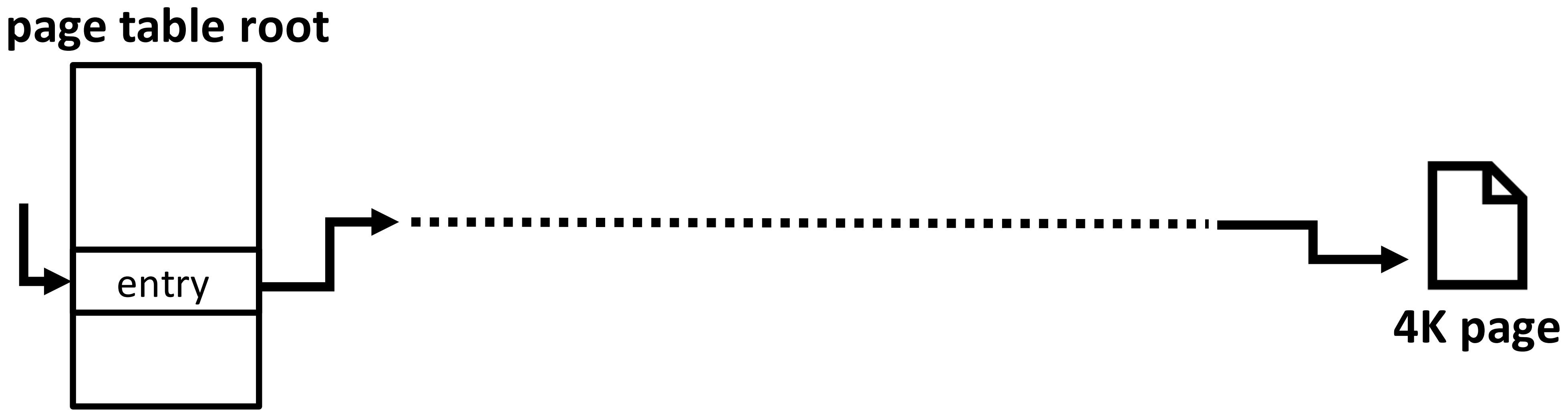
```
void *sbrk(intptr_t increment)
```

```
void *sbrk_one_page()
```



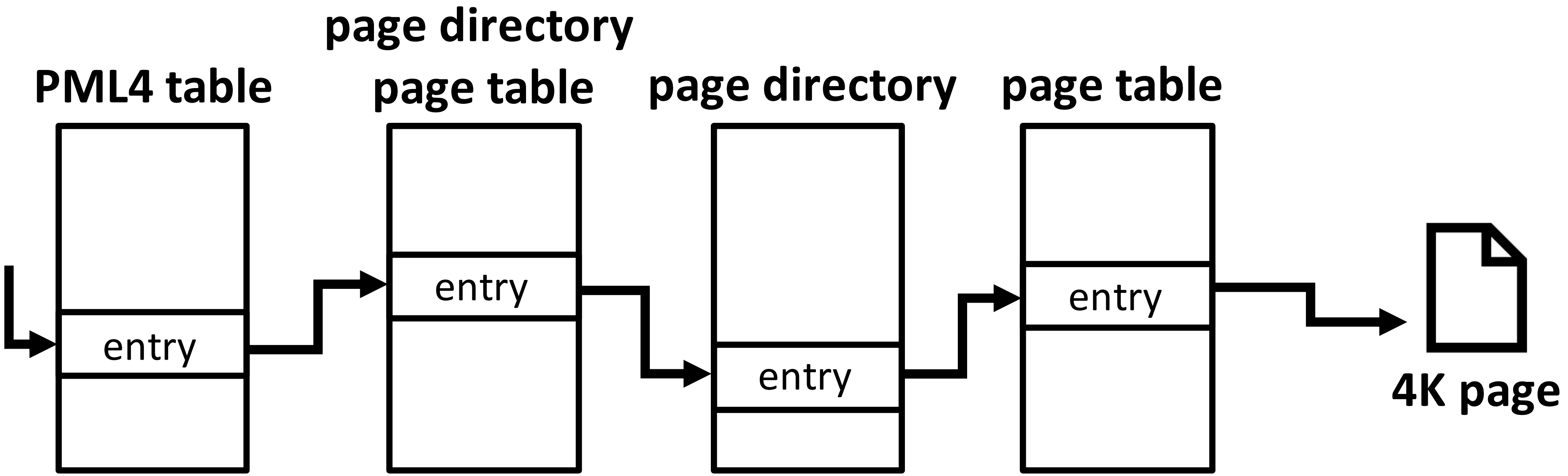
The sbrk() system call: Decomposition

```
void *sbrk_one_page()
```

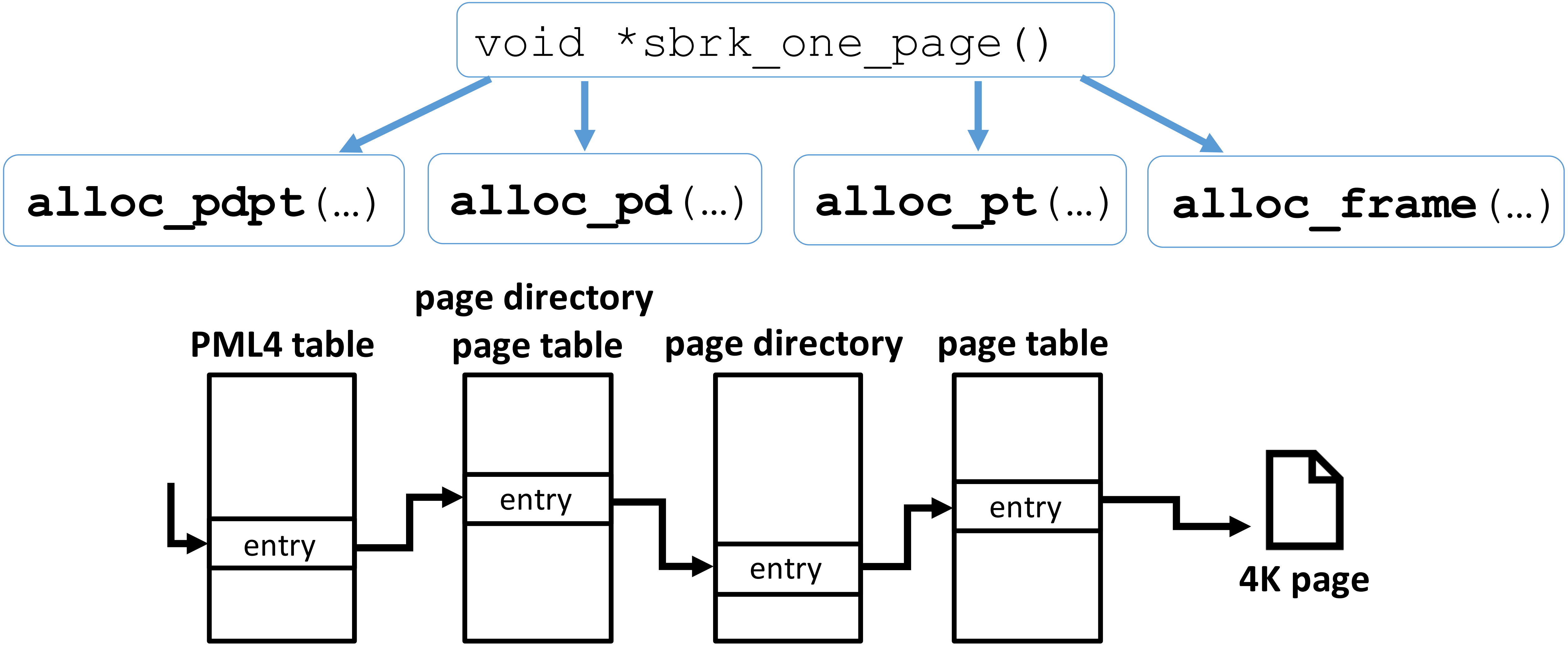


The sbrk() system call: Decomposition

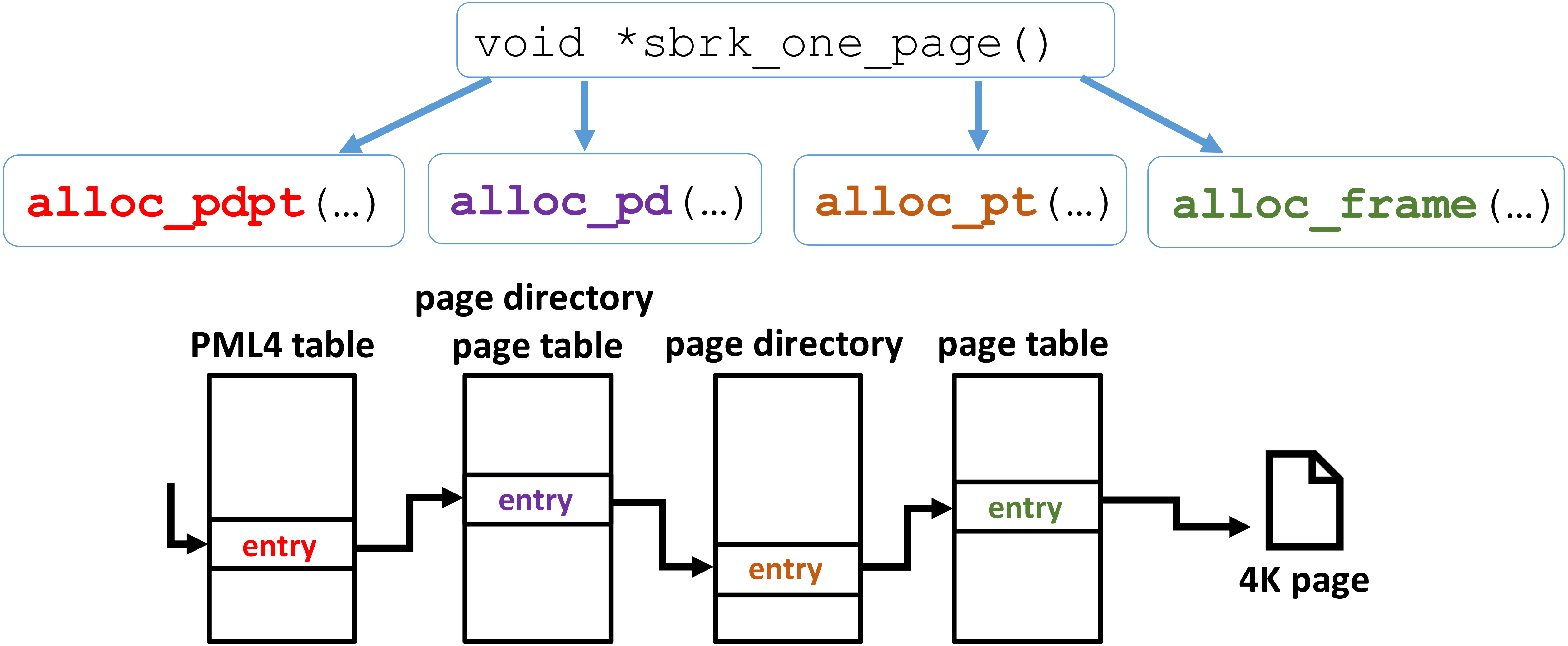
```
void *sbrk_one_page()
```



The sbrk() system call: Decomposition



The sbrk() system call: Decomposition



The sbrk() system call: Decomposition

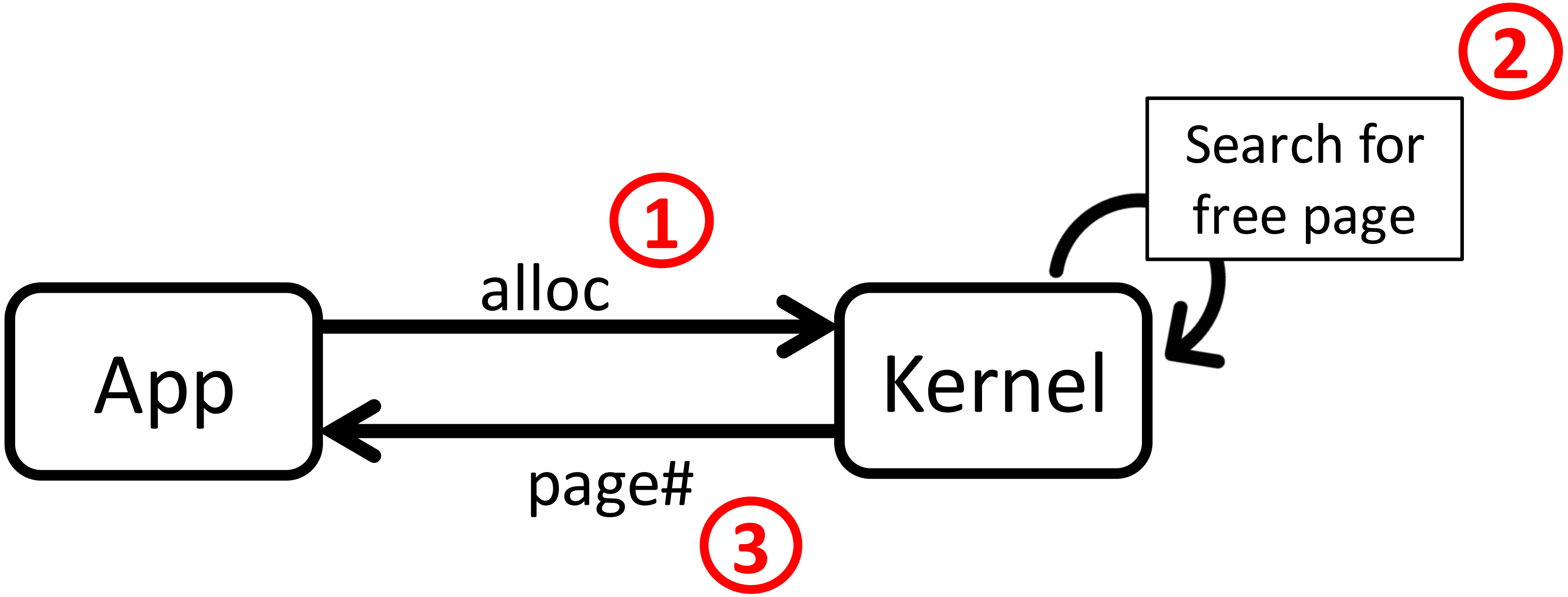
```
int alloc_pdpt(int pml4, size_t index)
```

```
int alloc_pd(int pdpt, size_t index)
```

```
int alloc_pt(int pd, size_t index)
```

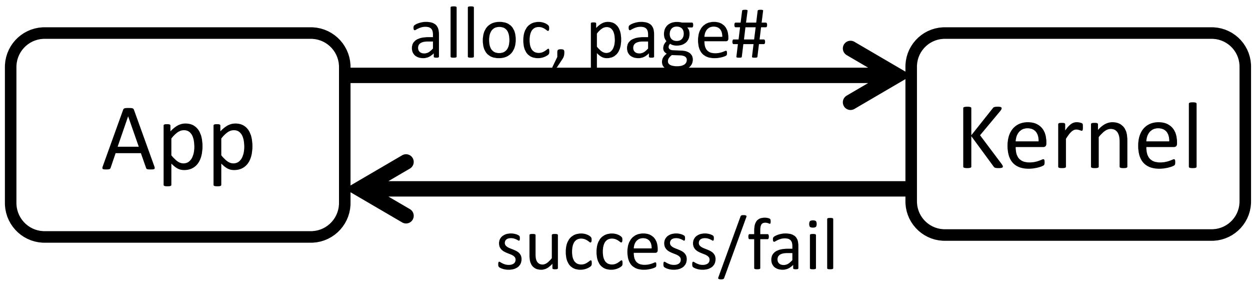
```
int alloc_frame(int pt, size_t index)
```

The sbrk() system call: Explicit allocation



The sbrk() system call: Explicit allocation

- Kernel keeps track of per-page metadata: owner/type
- User space searches for free page; kernel validates



The sbrk() system call: Finite Interface

```
int alloc_pdpt(int pml4, size_t index, int free_pn)
```

```
int alloc_pd(int pdpt, size_t index, int free_pn)
```

```
int alloc_pt(int pd, size_t index, int free_pn)
```

```
int alloc_frame(int pt, size_t index, int free_pn)
```

- Any composition of these system calls maintains **isolation**

> For any virtual address in a process p,
> if the virtual address maps to a page
> the page must be exclusively owned by p.

Implementation

Component	Lines	Languages
Kernel implementation	7,616	C, assembly
State-machine specification	804	Python
Declarative specification	263	Python
Verifier	2,878	C++, Python
User-space implementation	10,025	C, assembly

Outline

- Verification workflow
- Finite interface design
- **Demo**
- Evaluation & lessons learned

Demo

- Hyperkernel in action
- Catching a low-level bug producing a stack trace
- Catching a process isolation bug producing a visualized counterexample

Outline

- Verification workflow
- Finite interface design
- Demo
- **Evaluation & lessons learned**

What was the development effort?

- Write a state machine specification
- Relate LLVM data structures to abstract specification state
- Write checks for the representation invariants if needed.

What was the development effort?

- Write a state machine specification
- Relate LLVM data structures to abstract specification state
- Write checks for the representation invariants if needed.

- **Adding and verifying a system call usually takes < 1 hour**

Is the design effective for scalable verification?

- 45 minutes on a single core machine
- 15 minutes on an 8-core Intel i7
- Not sensitive to system parameters (e.g., number of pages)
- **Design is effective for scalable verification**

Conclusion

- Feasible to verify simple Unix-like OS kernel
- Automatic verification through symbolic execution
  - Make interface finite
  - Decompose complex system calls to scale verification
- Verifiability as a first-class system design concern
- http://locore.cs.washington.edu/hyperkernel