USENIX ATC '24

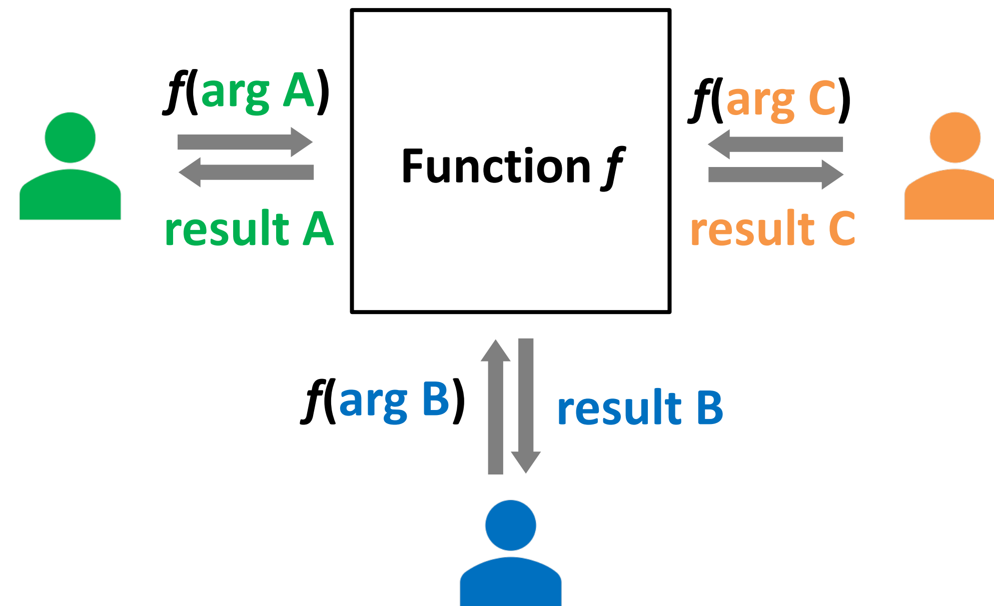
A Secure, Fast, and Resource-Efficient Serverless Platform with Function REWIND

Jaehyun Song[1], Bumsuk Kim[1], Minwoo Kwak[2], Byoungyoung Lee[3], Euiseong Seo[1], and Jinkyu Jeong[2]

Sungkyunkwan University[1]
Yonsei University[2]
Seoul National University[3]

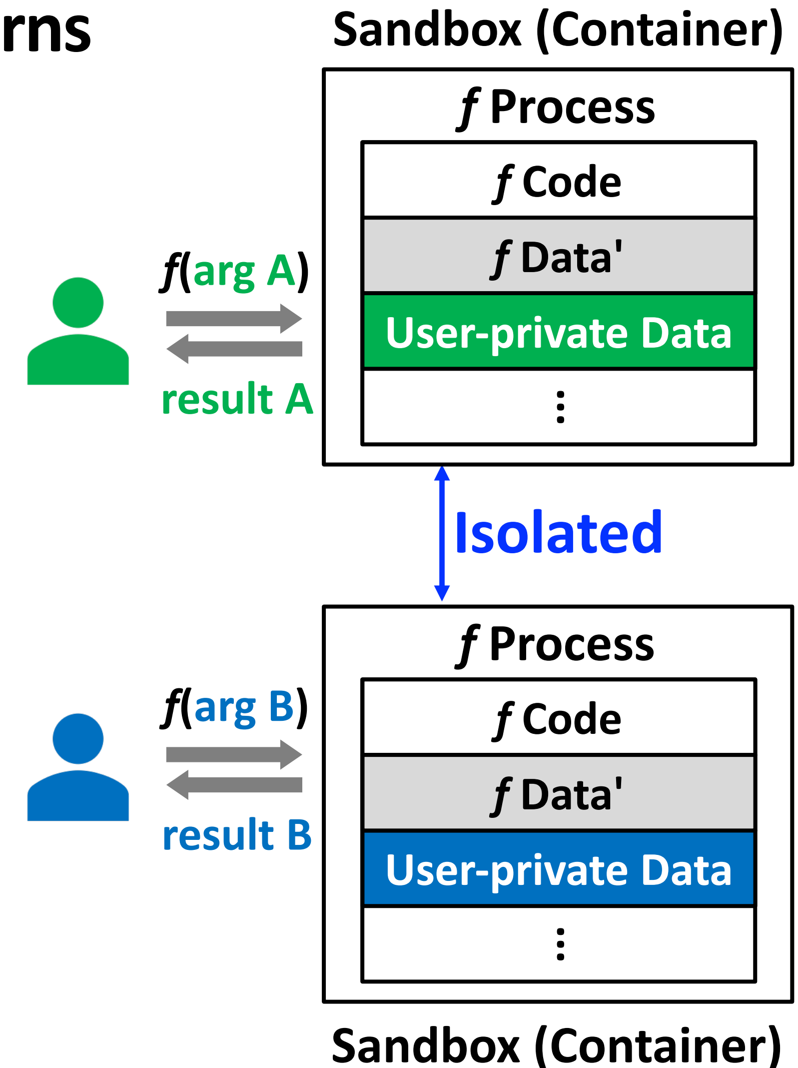
Serverless Computing

- **Serverless computing** has gained traction in **cloud** computing
  - Major cloud vendors adopted serverless computing
  - Developers write **functions**, each function handles requests from **multiple users**



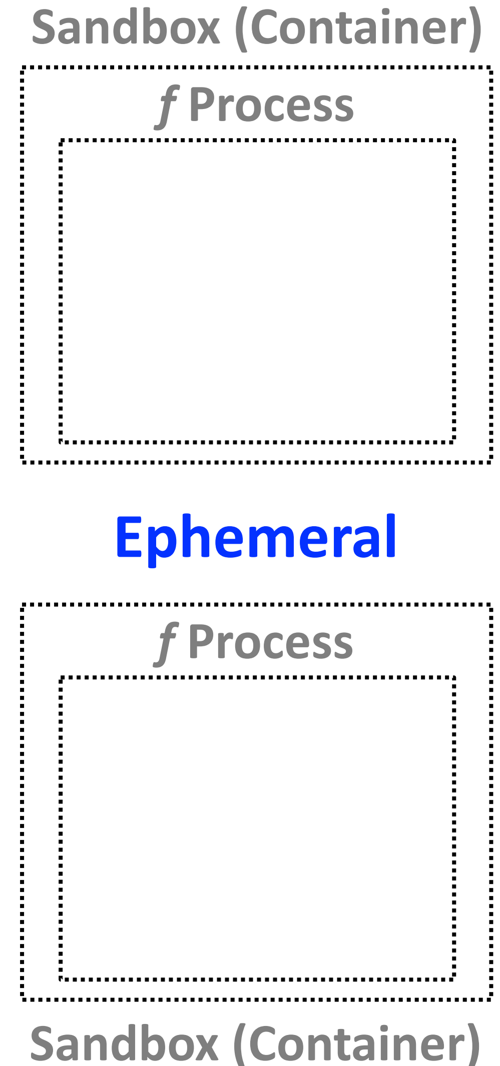
Security in Serverless Computing

- Original serverless computing has **no security concerns**
  - Functions are stateless
    - States of the function disappears after execution
  - Functions run in an ephemeral sandbox
    - **Sandbox** (i.e., container) provides isolation



Security in Serverless Computing

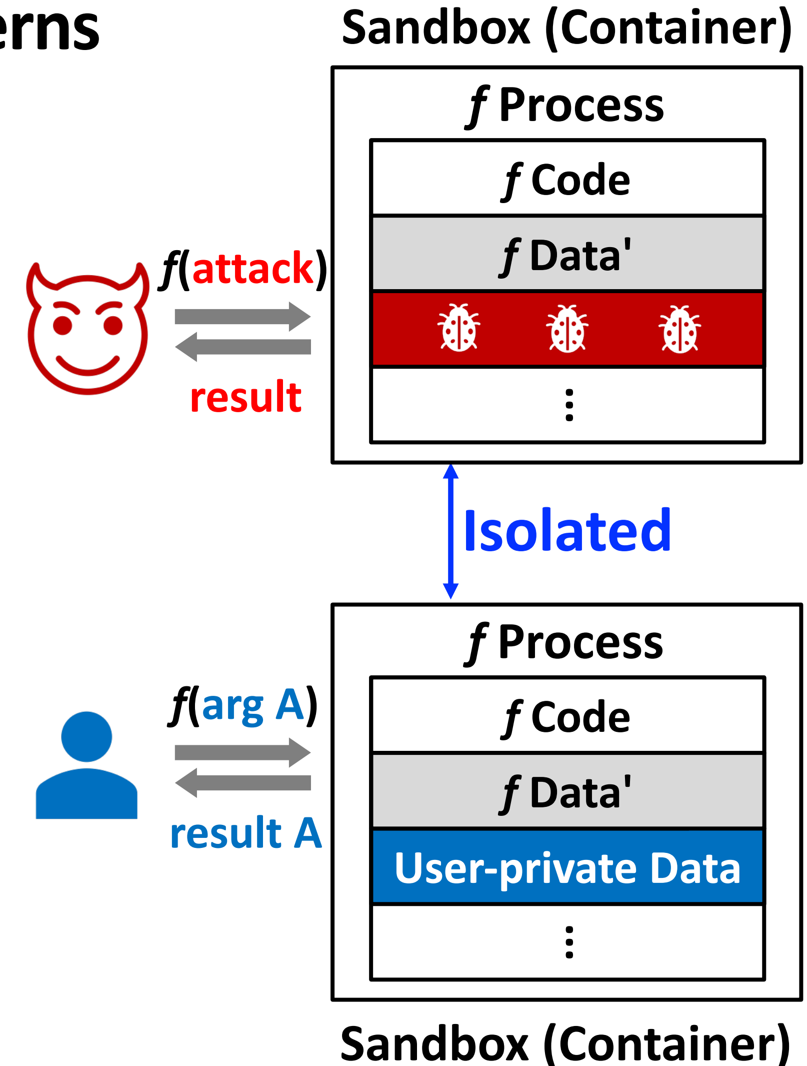
- Original serverless computing has **no security concerns**
  - Functions are stateless
    - States of the function disappears after execution
  - Functions run in an ephemeral sandbox
    - **Sandbox** (i.e., container) provides isolation
    - **Ephemeral** sandbox eliminates persistence of any data



Security in Serverless Computing

- Original serverless computing has **no security concerns**
  - Functions are stateless
    - States of the function disappears after execution
  - Functions run in an ephemeral sandbox
    - **Sandbox** (i.e., container) provides isolation
    - **Ephemeral** sandbox eliminates persistence of any data

- Cold-start overhead degrades performance

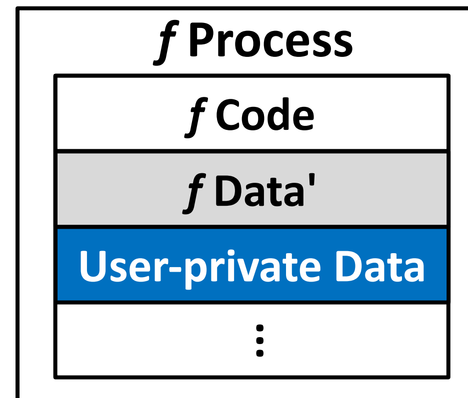


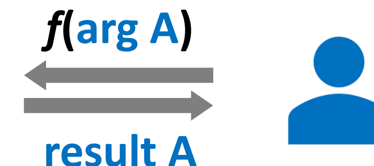
Container Reuse in Serverless Computing

- Container reuse is a prevalent technique to mitigate the cold-start overhead
- However, container reuse raises a security problem
  - **Quasi-persistence** [1, 2] of data
  - Attack opportunities of data exfiltration, rootkit, etc.

```
from code import func
...
do
  args = recv()
  result = func(args)
  send(result)
while keepalive == True
...
```

Function Handler





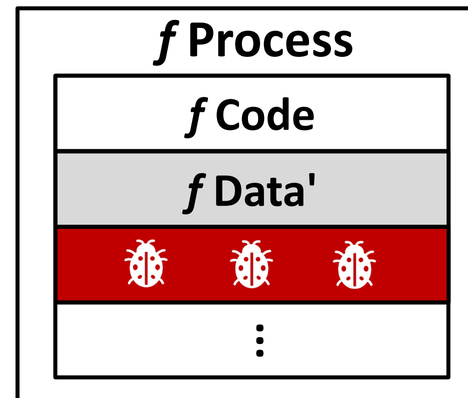
[1] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in faas. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 398–415, 2023.
[2] Datta, Pubali, et al. "{ALASTOR}: Reconstructing the provenance of serverless intrusions." *31st USENIX Security Symposium (USENIX Security 22)*. 2022.

Container Reuse in Serverless Computing

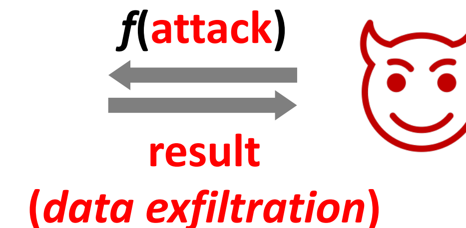
- Container reuse is a prevalent technique to mitigate the cold-start overhead
- However, container reuse raises a security problem
  - **Quasi-persistence** [1, 2] of data
  - Attack opportunities of data exfiltration, rootkit, etc.

```
from code import func
...
do
  args = recv()
  result = func(args)
  send(result)
while keepalive == True
...
```

Function Handler



Container



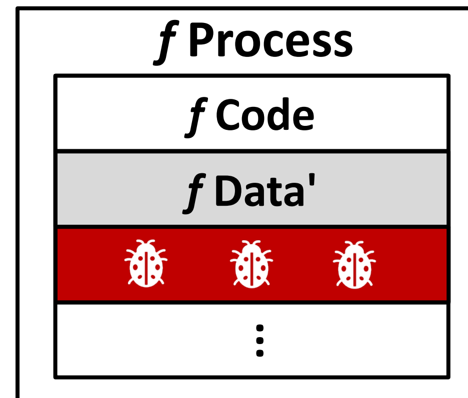
[1] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in faas. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 398–415, 2023.
[2] Datta, Pubali, et al. "{ALASTOR}: Reconstructing the provenance of serverless intrusions." *31st USENIX Security Symposium (USENIX Security 22)*. 2022.

Container Reuse in Serverless Computing

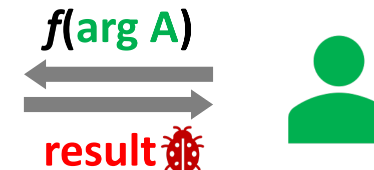
- Container reuse is a prevalent technique to mitigate the cold-start overhead
- However, container reuse raises a security problem
  - **Quasi-persistence** [1, 2] of data
  - Attack opportunities of data exfiltration, rootkit, etc.

```
from code import func
...
do
  args = recv()
  result = func(args)
  send(result)
while keepalive == True
...
```

Function Handler



Container



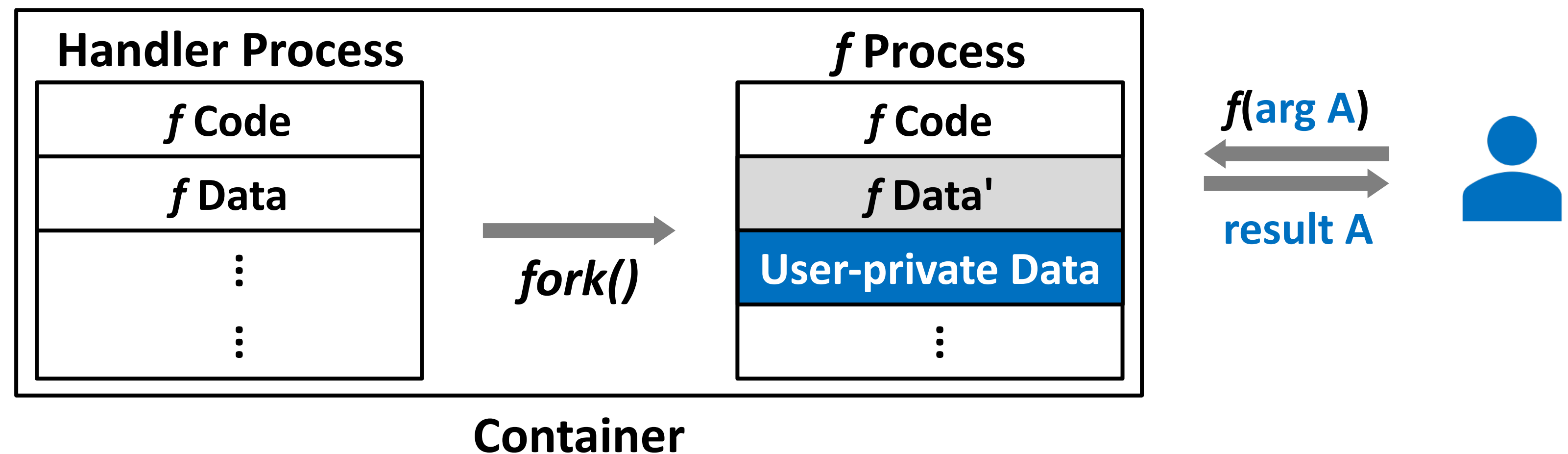
[1] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in faas. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 398–415, 2023.
[2] Datta, Pubali, et al. "{ALASTOR}: Reconstructing the provenance of serverless intrusions." *31st USENIX Security Symposium (USENIX Security 22)*. 2022.

Alleviating Security Issues #1 – Fork

- *fork()* removes **memory persistence** through process isolation for each request
  - The function handler process forks a child process to handle each function request

```
from code import func
...
do
  args = recv()
  child = fork()
  if (child == 0):
    result = func(args)
    exit(result)
  else:
    wait(child, &result)
  send(result)
while keepalive == True
...
```

Function Handler

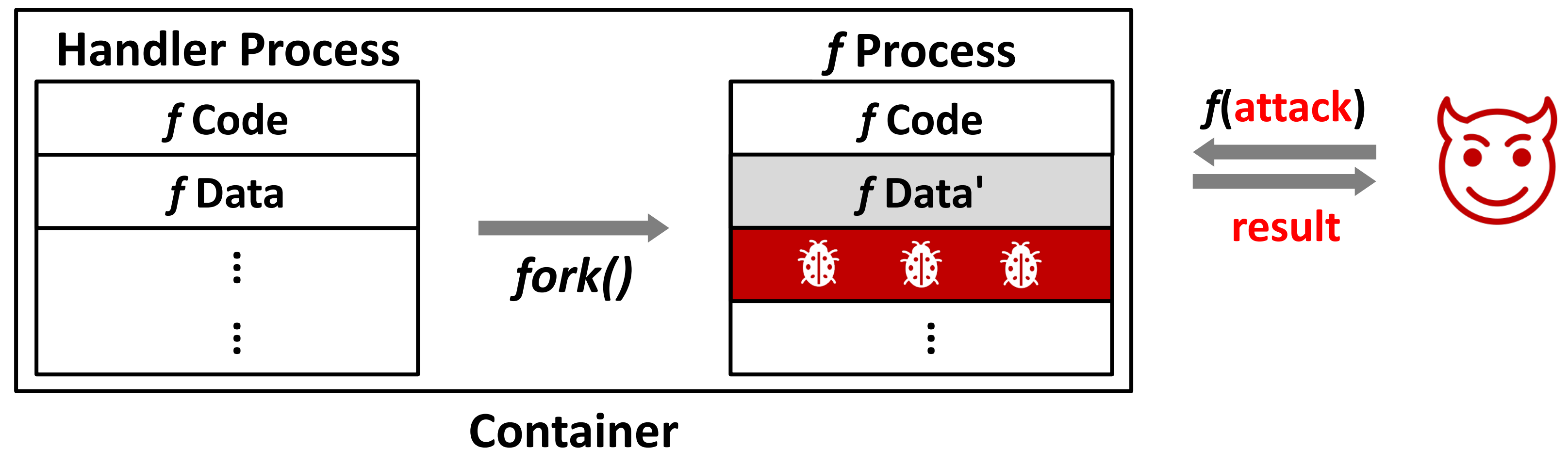


Alleviating Security Issues #1 – Fork

- *fork()* removes **memory persistence** through process isolation for each request
  - The function handler process forks a child process to handle each function request

```
from code import func
...
do
  args = recv()
  child = fork()
  if (child == 0):
    result = func(args)
    exit(result)
  else:
    wait(child, &result)
  send(result)
while keepalive == True
...
```

Function Handler



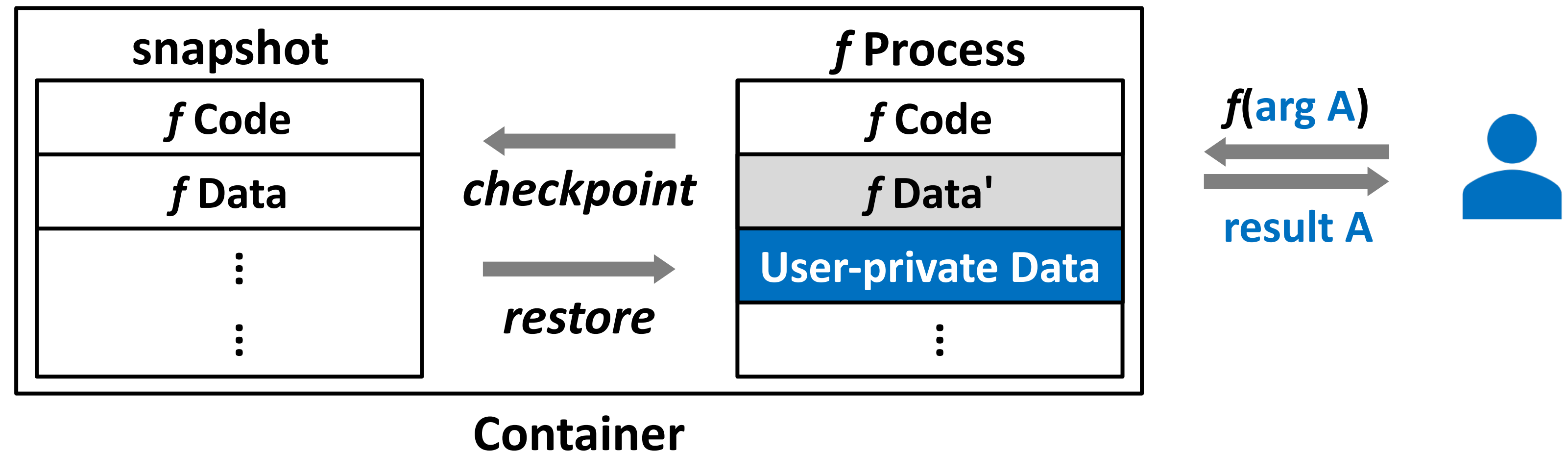
No attacks made effective!

Alleviating Security Issues #2 – Checkpoint/Restore

- Groundhog (GH) [1] removes **memory persistence** by using checkpoint/restore
  - Checkpoint a function handler process before handling any function request
  - Restore a function handler process to its initial state after handling a function request

```
from code import func
...
do
  checkpoint()
  args = recv()
  result = func(args)
  send(result)
  restore()
while keepalive == True
...
```

Function Handler



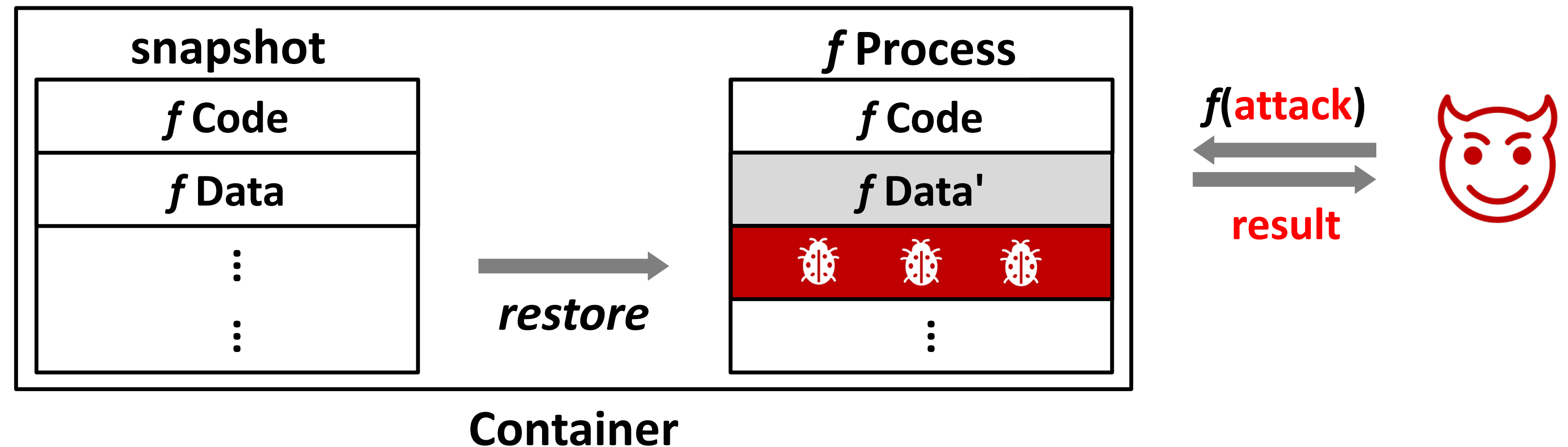
[1] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in faas. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 398–415, 2023.

Alleviating Security Issues #2 – Checkpoint/Restore

- Groundhog (GH) [1] removes **memory persistence** by using checkpoint/restore
  - Checkpoint a function handler process before handling any function request
  - Restore a function handler process to its initial state after handling a function request

```
from code import func
...
do
  checkpoint()
  args = recv()
  result = func(args)
  send(result)
  restore()
while keepalive == True
...
```

Function Handler



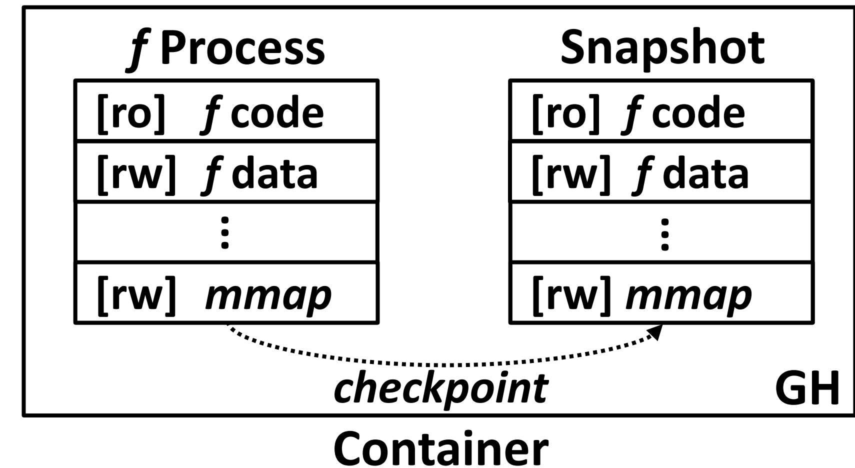
No attacks made effective!

[1] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in faas. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 398–415, 2023.

Problems of Previous Approaches

- No consideration for the **repeated execution** of function request

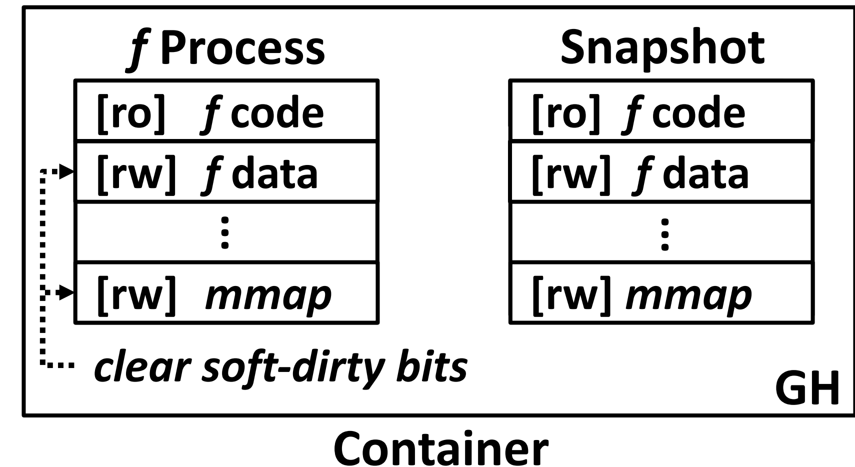
- **Problem #1**: **memory space** overhead
  - GH copies all data to the snapshot to recover initial state
  - The repeated execution allows further optimization opportunities



Problems of Previous Approaches

- No consideration for the **repeated execution** of function request

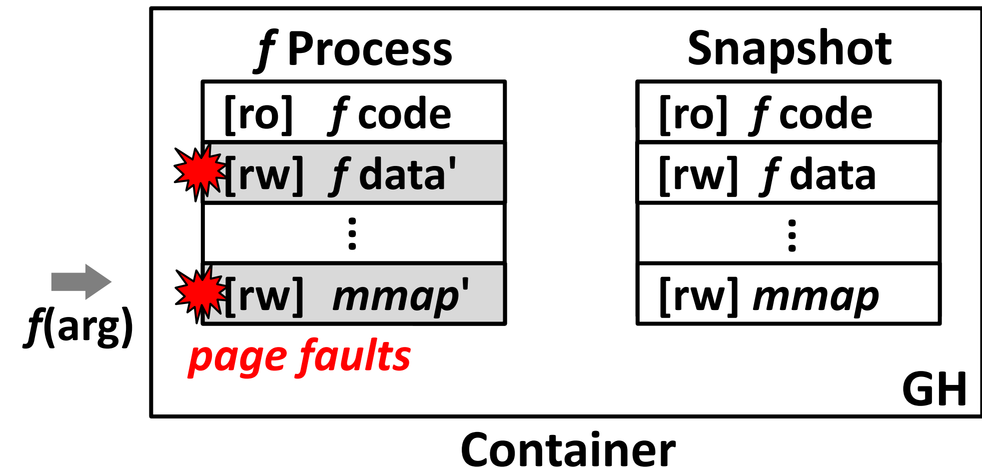
- **Problem #2**: **repeated page fault** overheads
  - GH recovers modified data after request handling
    - Tracking modified data requires page faults (Linux's soft-dirty feature)



Problems of Previous Approaches

- No consideration for the **repeated execution** of function request

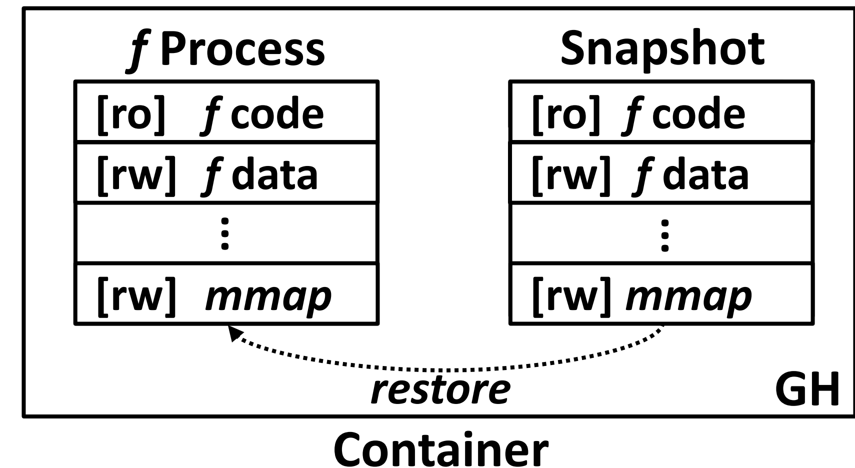
- **Problem #2**: **repeated page fault** overheads
  - GH recovers modified data after request handling
    - Tracking modified data requires page faults (Linux's soft-dirty feature)



Problems of Previous Approaches

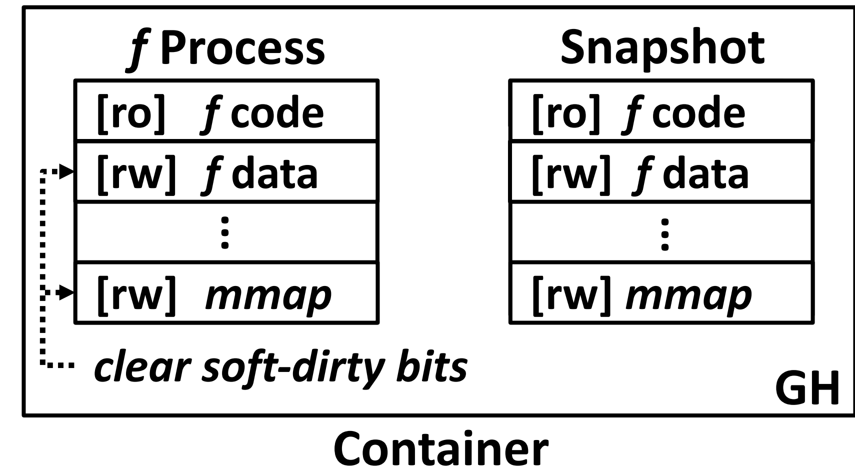
- No consideration for the **repeated execution** of function request

- **Problem #2**: **repeated page fault** overheads
  - GH recovers modified data after request handling
    - Tracking modified data requires page faults
      (Linux's soft-dirty feature)



Problems of Previous Approaches

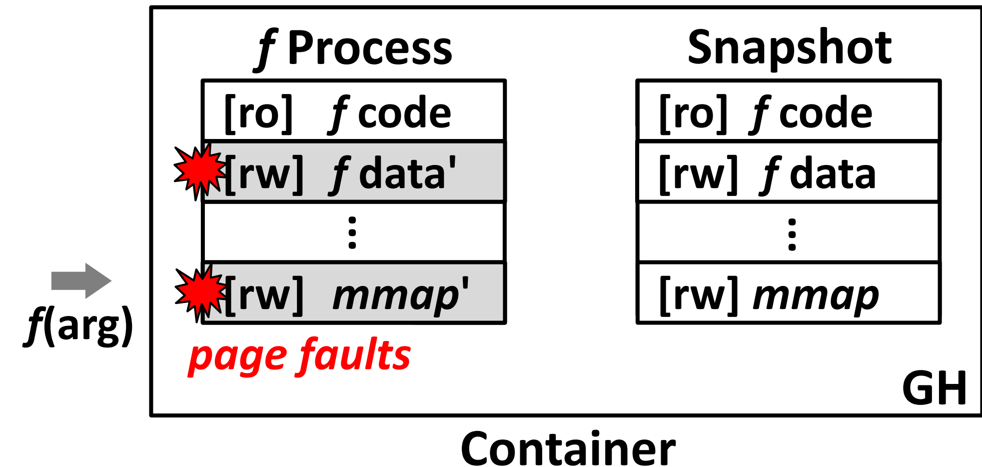
- No consideration for the **repeated execution** of function request
- **Problem #2**: **repeated page fault** overheads
  - GH recovers modified data after request handling
    - Tracking modified data requires page faults (Linux's soft-dirty feature)



Problems of Previous Approaches

- No consideration for the **repeated execution** of function request

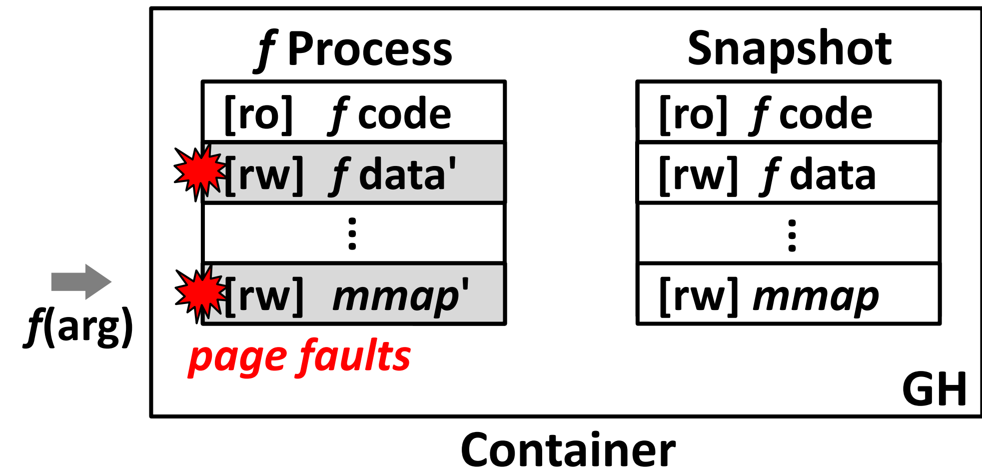
- **Problem #2**: **repeated page fault** overheads
  - GH recovers modified data after request handling
    - Tracking modified data requires page faults (Linux's soft-dirty feature)

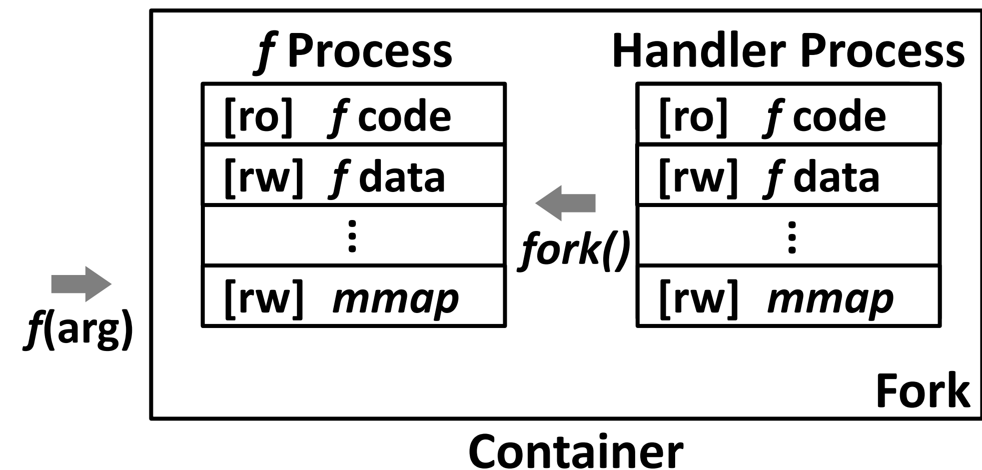


Problems of Previous Approaches

- No consideration for the **repeated execution** of function request

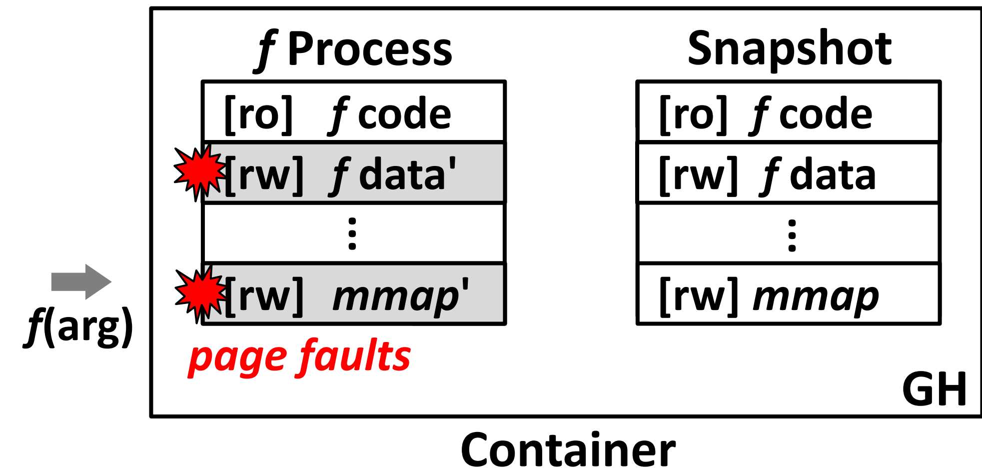
- **Problem #2**: **repeated page fault** overheads
  - GH recovers modified data after request handling
    - Tracking modified data requires page faults (Linux's soft-dirty feature)
  - *fork*() causes copy-on-write page faults for modified data

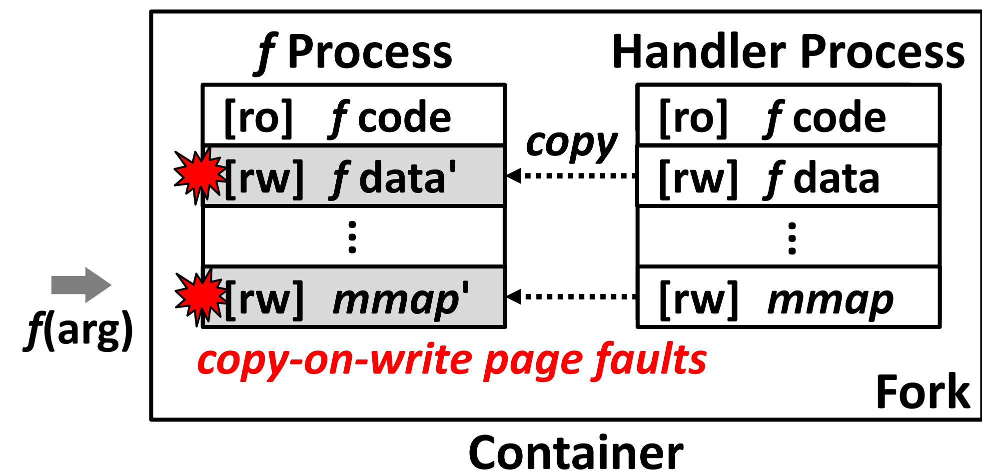




Problems of Previous Approaches

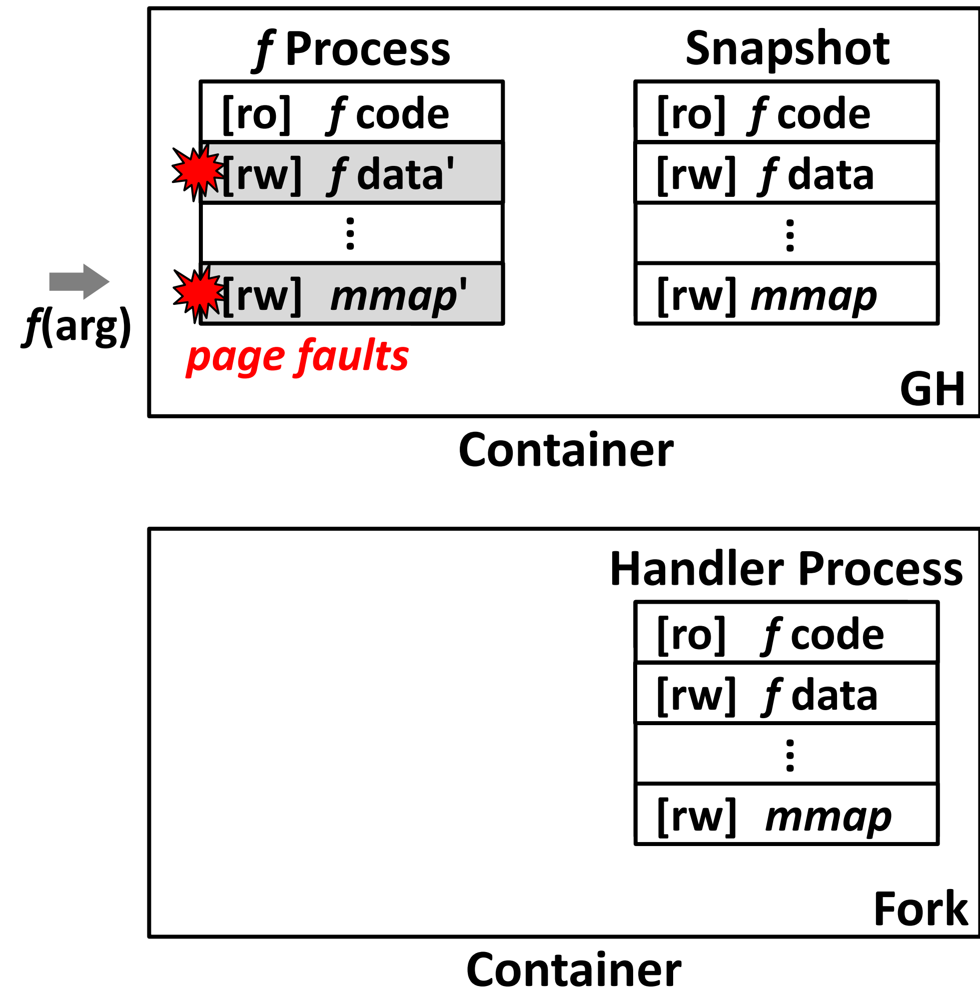
- No consideration for the **repeated execution** of function request
- **Problem #2**: **repeated page fault** overheads
  - GH recovers modified data after request handling
    - Tracking modified data requires page faults (Linux's soft-dirty feature)
  - *fork*() causes copy-on-write page faults for modified data





Problems of Previous Approaches

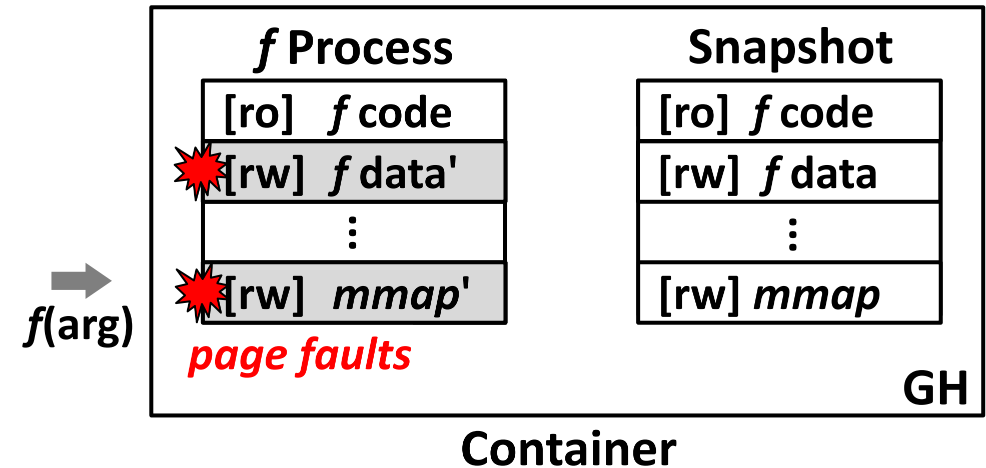
- No consideration for the **repeated execution** of function request
- **Problem #2**: **repeated page fault** overheads
  - GH recovers modified data after request handling
    - Tracking modified data requires page faults (Linux's soft-dirty feature)
  - *fork*() causes copy-on-write page faults for modified data

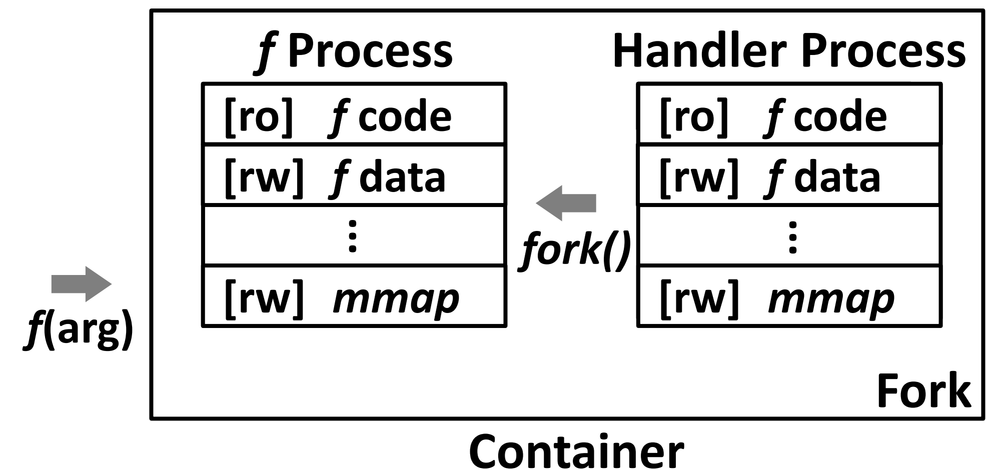


Problems of Previous Approaches

- No consideration for the **repeated execution** of function request

- **Problem #2**: **repeated page fault** overheads
  - GH recovers modified data after request handling
    - Tracking modified data requires page faults (Linux's soft-dirty feature)
  - *fork*() causes copy-on-write page faults for modified data

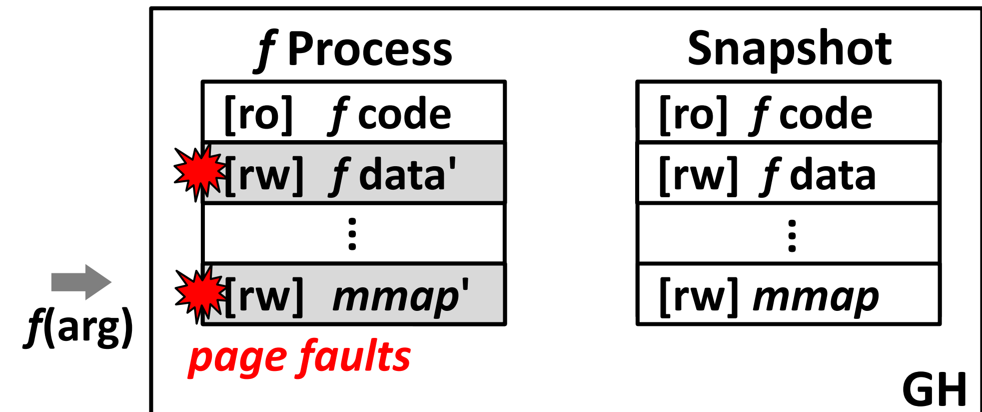


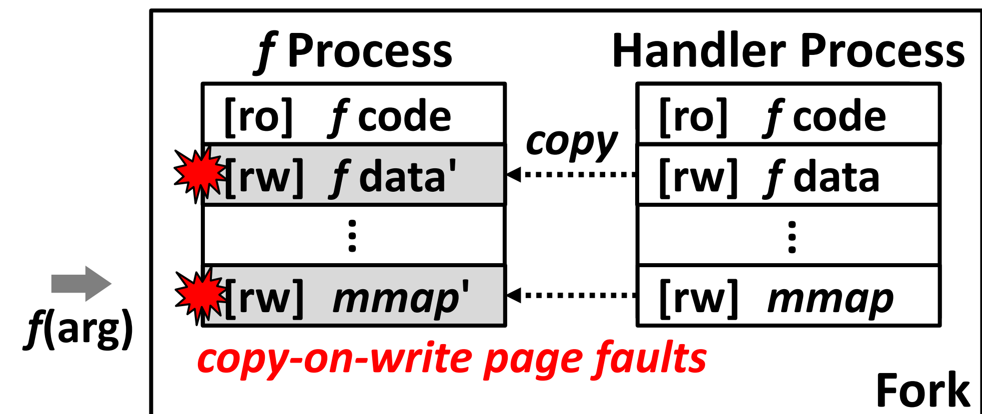


Problems of Previous Approaches

- No consideration for the **repeated execution** of function request

- **Problem #2**: **repeated page fault** overheads
  - GH recovers modified data after request handling
    - Tracking modified data requires page faults (Linux's soft-dirty feature)
  - *fork*() causes copy-on-write page faults for modified data

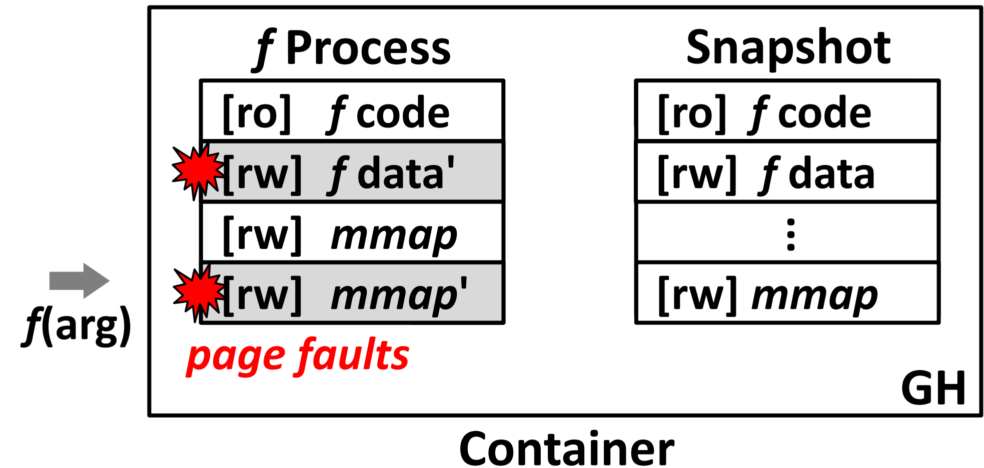


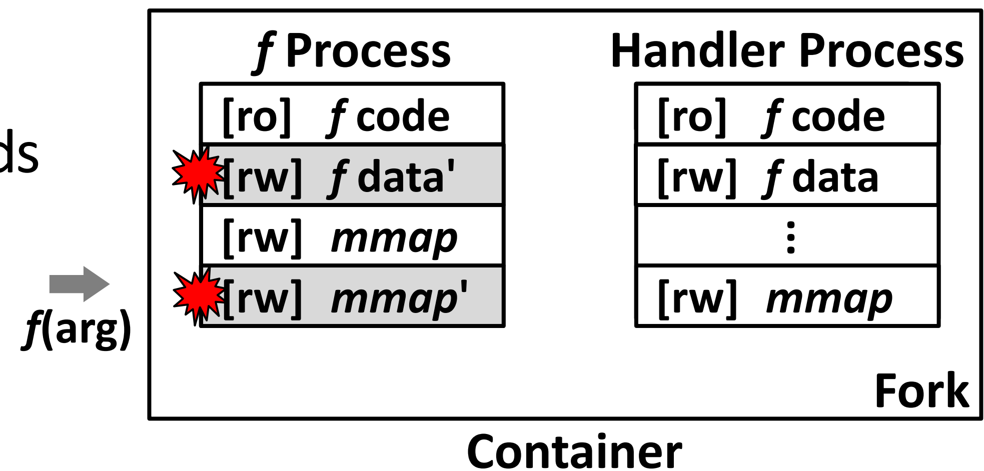


Problems of Previous Approaches

- No consideration for the **repeated execution** of function request

- **Problem #2**: **repeated page fault** overheads
  - GH recovers modified data after request handling
    - Tracking modified data requires page faults (Linux's soft-dirty feature)
  - *fork*() causes copy-on-write page faults for modified data
  - New *mmap*()s after snapshot/fork causes overheads (page allocation + page faults)
  - →These overheads repeat on <u>every function request handling</u>

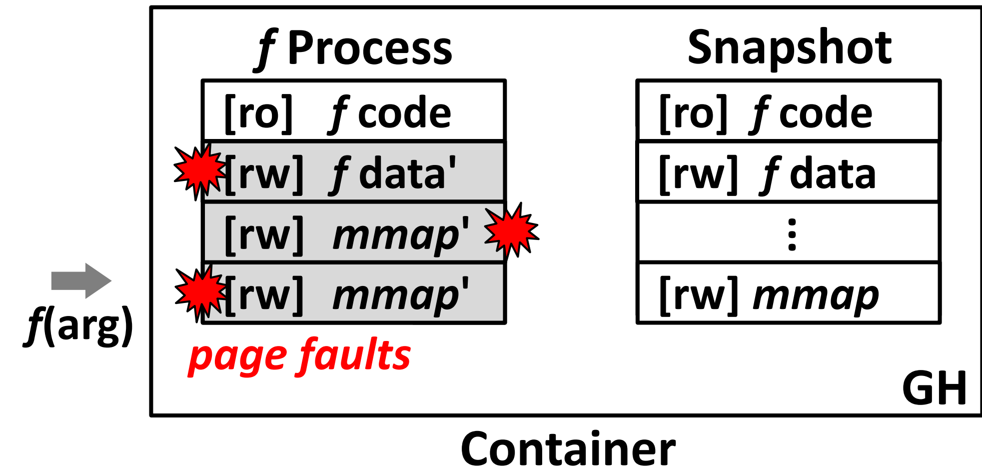


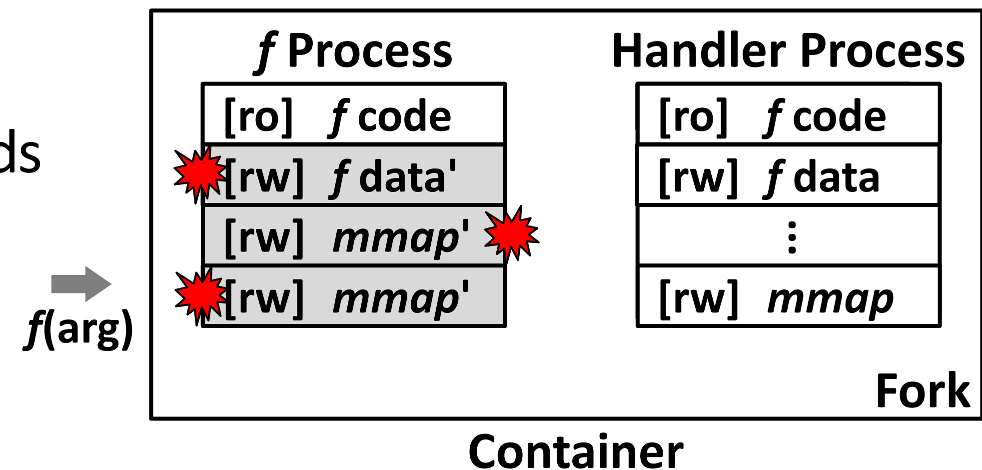


Problems of Previous Approaches

- No consideration for the **repeated execution** of function request
- **Problem #2**: **repeated page fault** overheads
  - GH recovers modified data after request handling
    - Tracking modified data requires page faults (Linux's soft-dirty feature)
  - *fork*() causes copy-on-write page faults for modified data
  - New *mmap*()s after snapshot/fork causes overheads (page allocation + page faults)

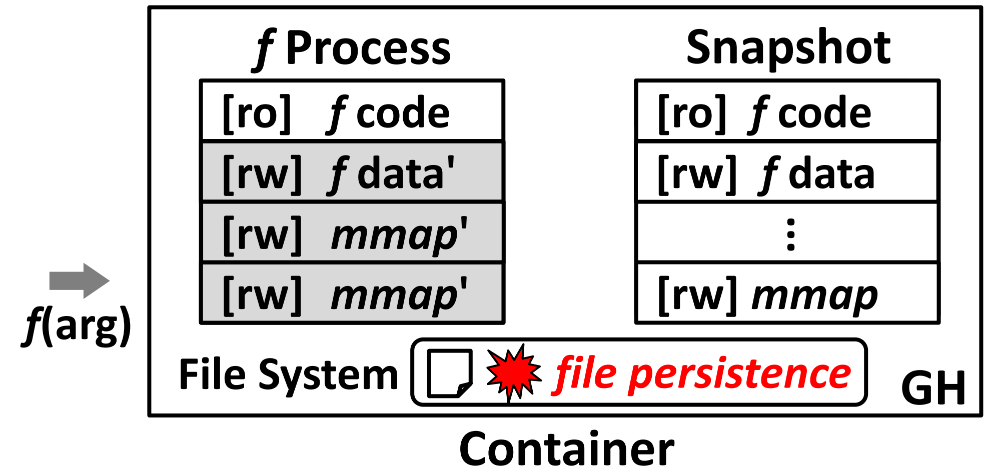
  → These overheads repeat on <u>every function request handling</u>

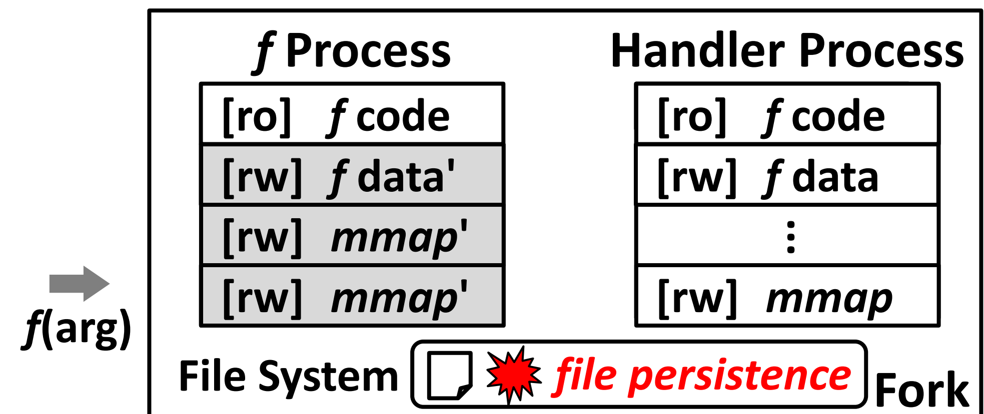




Problems of Previous Approaches

- **Problem #3**: no consideration of **file persistence**
  - Both schemes leave files after function executions
  - Files can contain privacy-sensitive data
  - Remaining files can be leaked or maliciously used





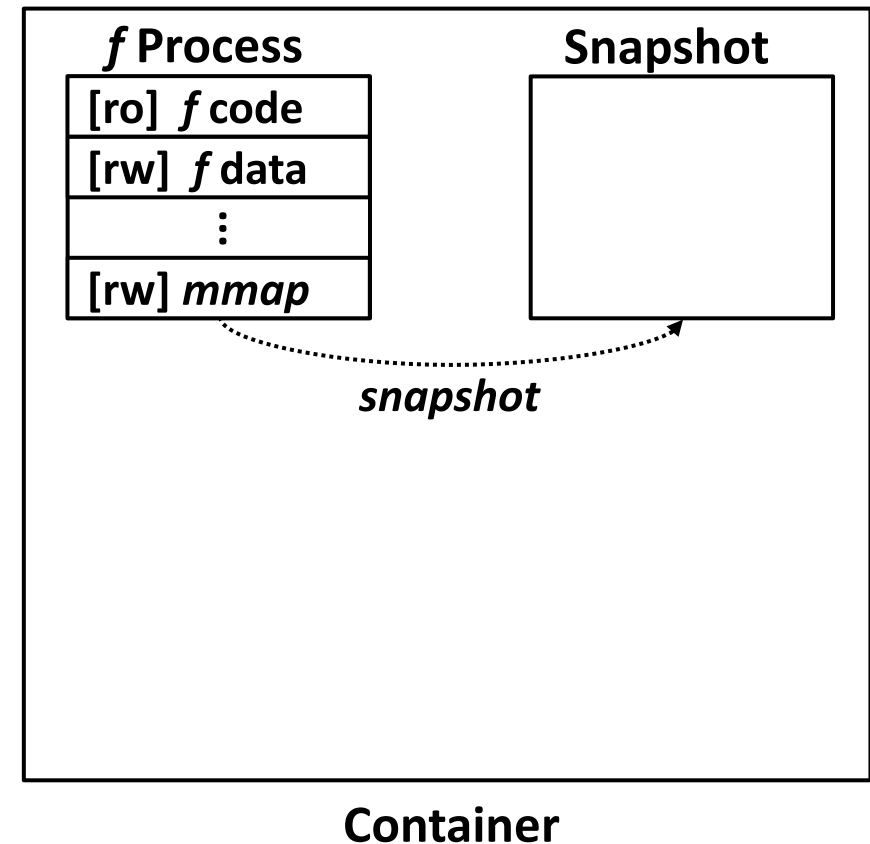
Our Approach: REWIND

- **Goal**: performance and memory efficient snapshot/restore
  - Elimination of memory and file persistence
  - Minimize memory usage for snapshot and reduce page faults
  - Key idea: **exploiting repeated handling** of function requests

- **Challenges:**
  - How does REWIND put only the original data of dirty pages to the snapshot?
  - How does REWIND track pages to dirty without page faults?

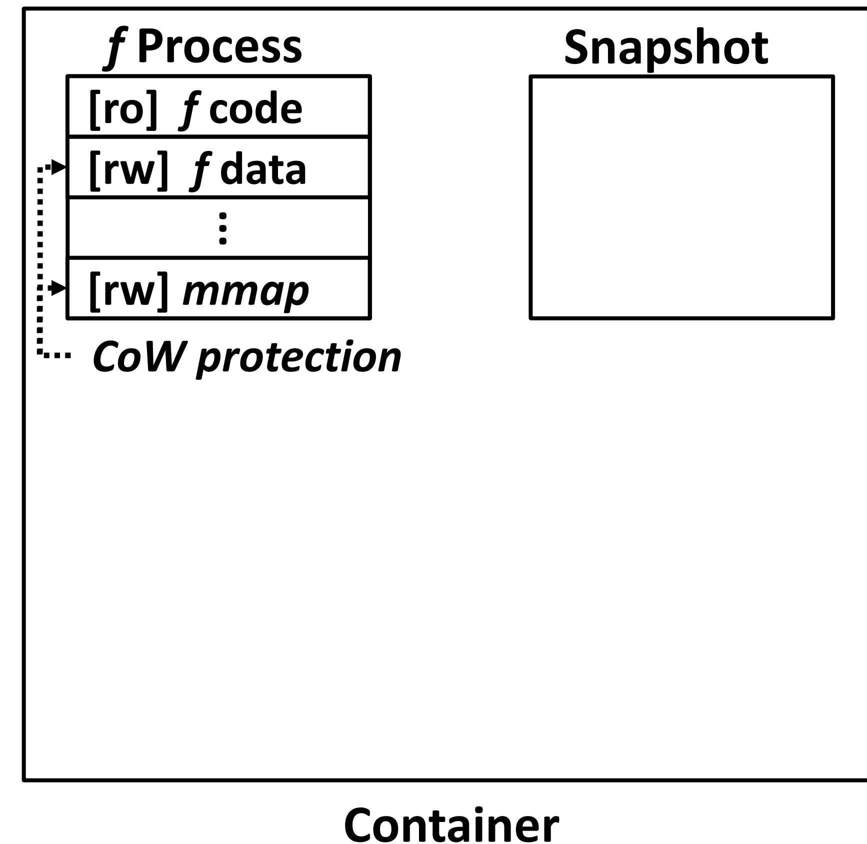
Snapshot/Rewind Operations

- ***snapshot***()
  - Take a snapshot of each page only when a page is about to be dirtied
    - Copy-on-write protection + *buddy page table*



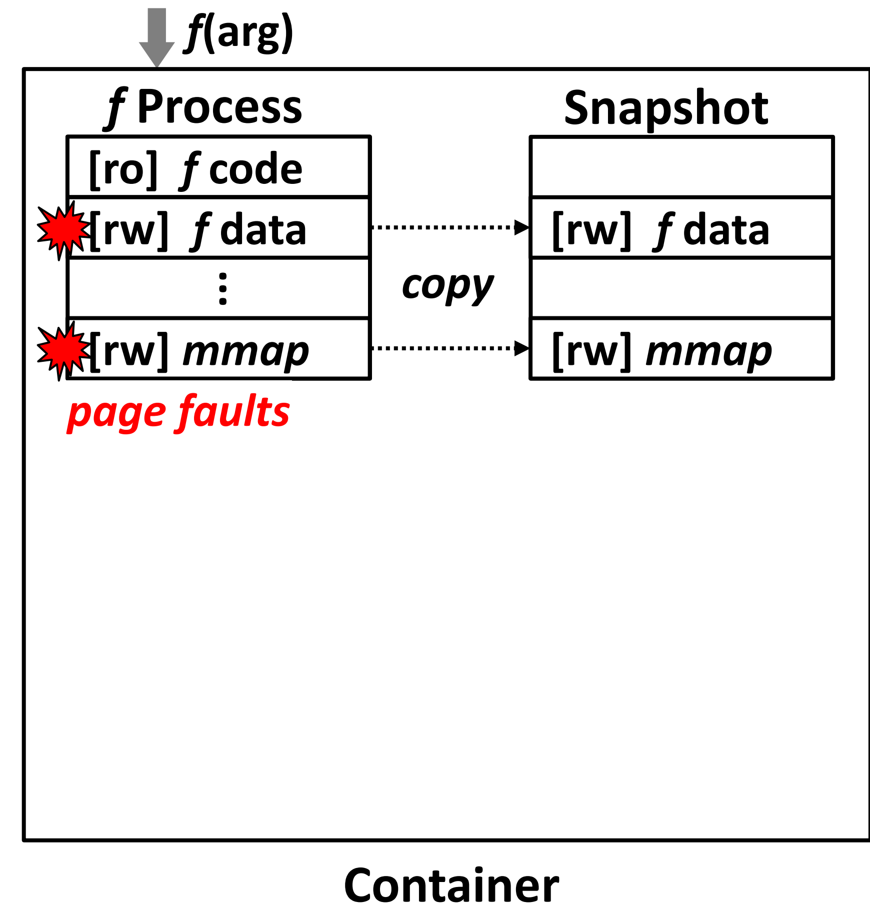
Snapshot/Rewind Operations

- ***snapshot***()
  - Take a snapshot of each page only when a page is about to be dirtied
    - Copy-on-write protection + *buddy page table*



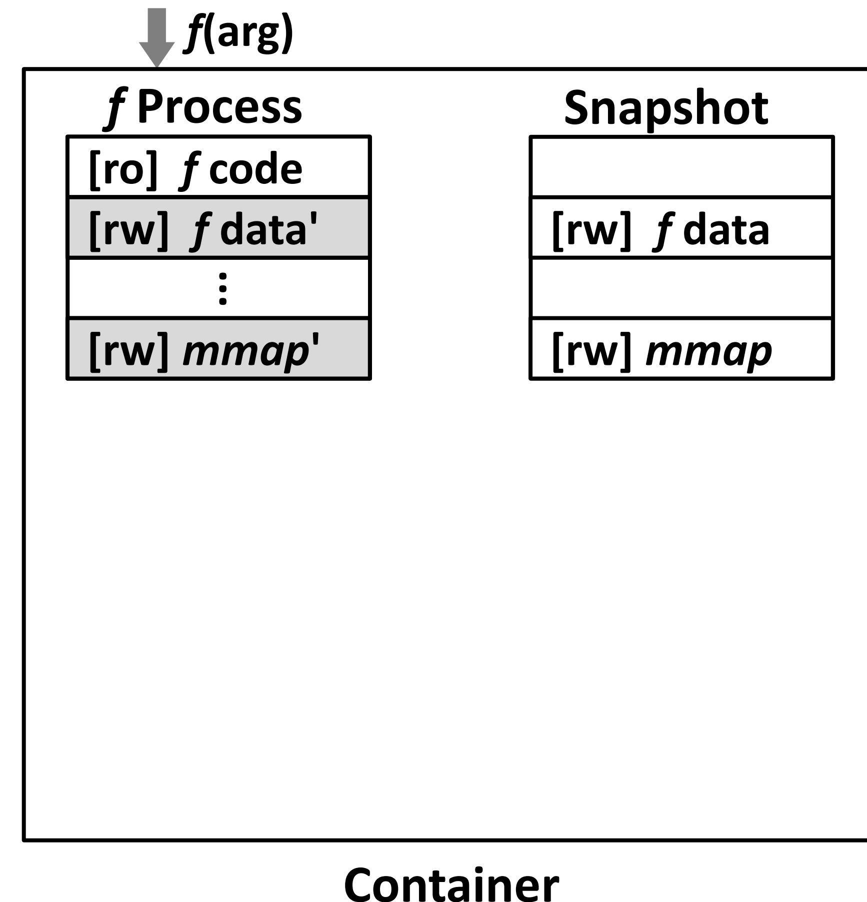
Snapshot/Rewind Operations

- ***snapshot***()
  - Take a snapshot of each page only when a page is about to be dirtied
    - Copy-on-write protection + *buddy page table*



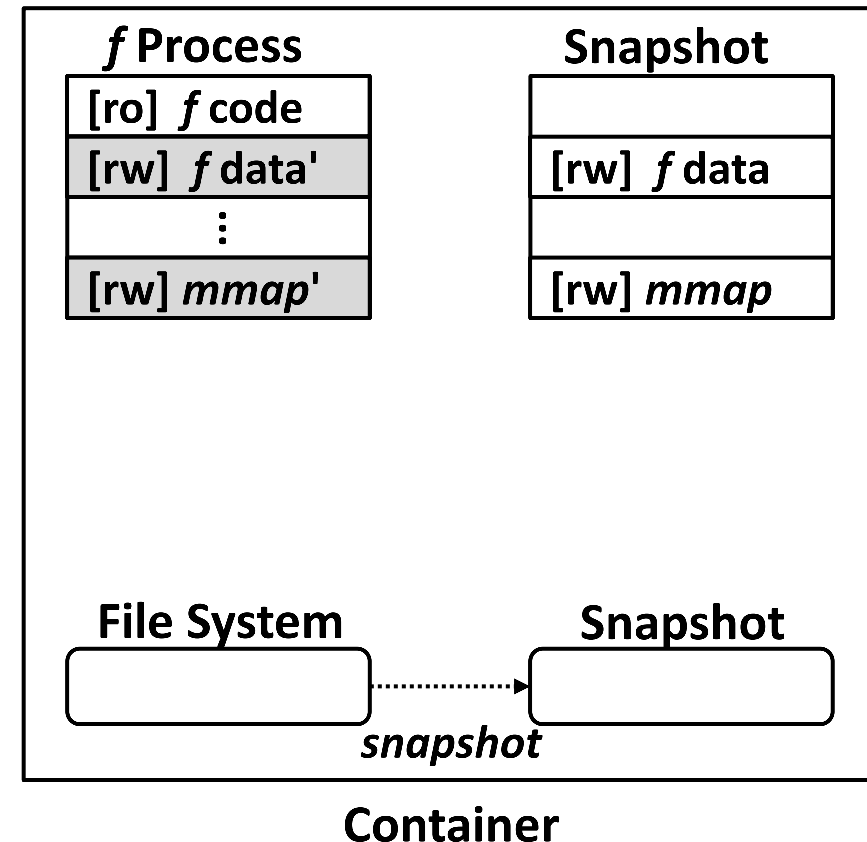
Snapshot/Rewind Operations

- ***snapshot***()
  - Take a snapshot of each page only when a page is about to be dirtied
    - Copy-on-write protection + *buddy page table*
  - For repeated dirty pages, keep pages duplicated (snapshot + original)
  - For zero pages, do NOT maintain in a snapshot to save memory



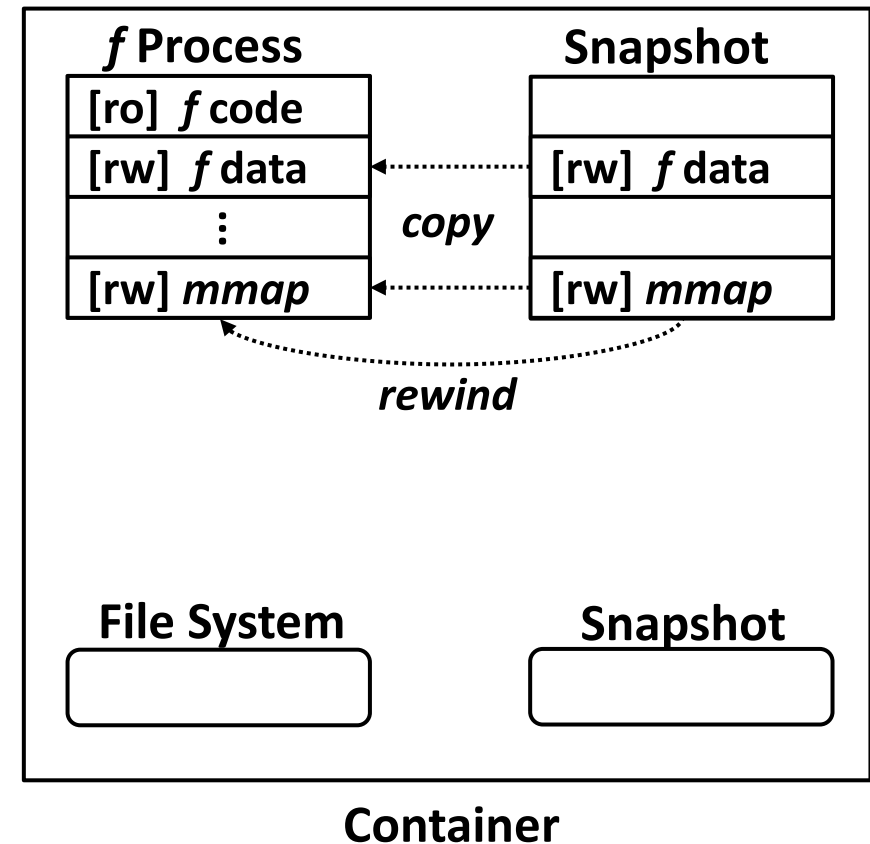
Snapshot/Rewind Operations

- ***snapshot***()
  - Take a snapshot of each page only when a page is about to be dirtied
    - Copy-on-write protection + *buddy page table*
  - For repeated dirty pages, keep pages duplicated (snapshot + original)
  - For zero pages, do NOT maintain in a snapshot to save memory
  - Keep snapshot of files



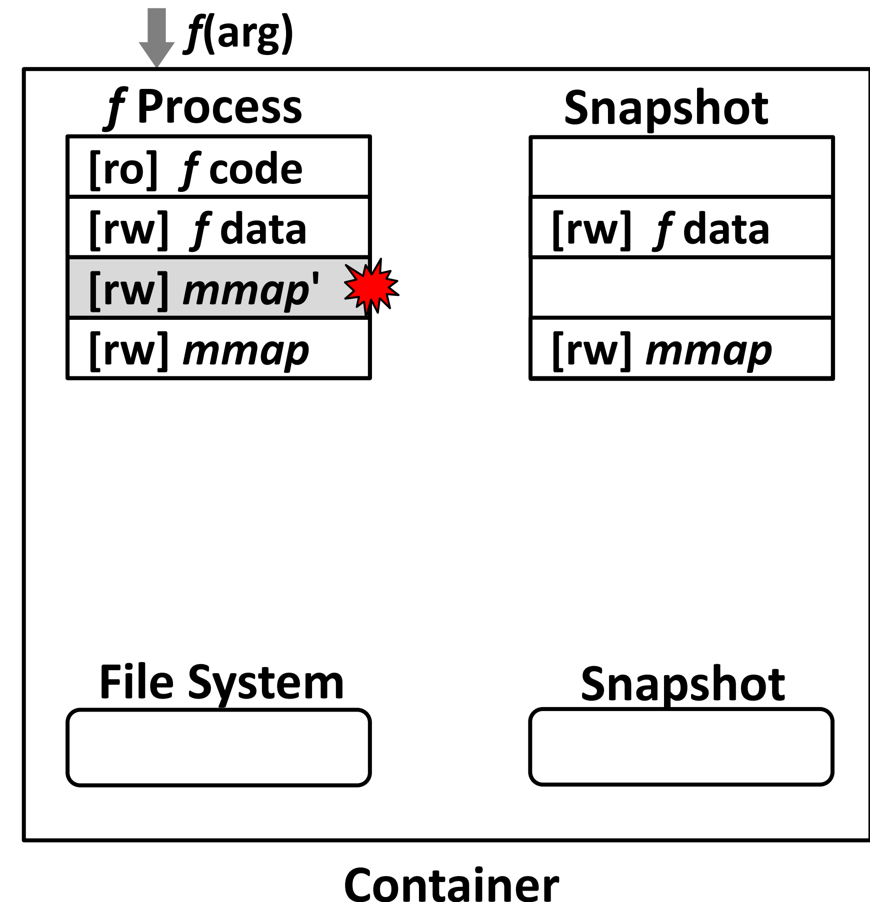
Snapshot/Rewind Operations

- ***rewind***()
  - Restore dirty pages to original ones
    - Reset pages to zero if necessary



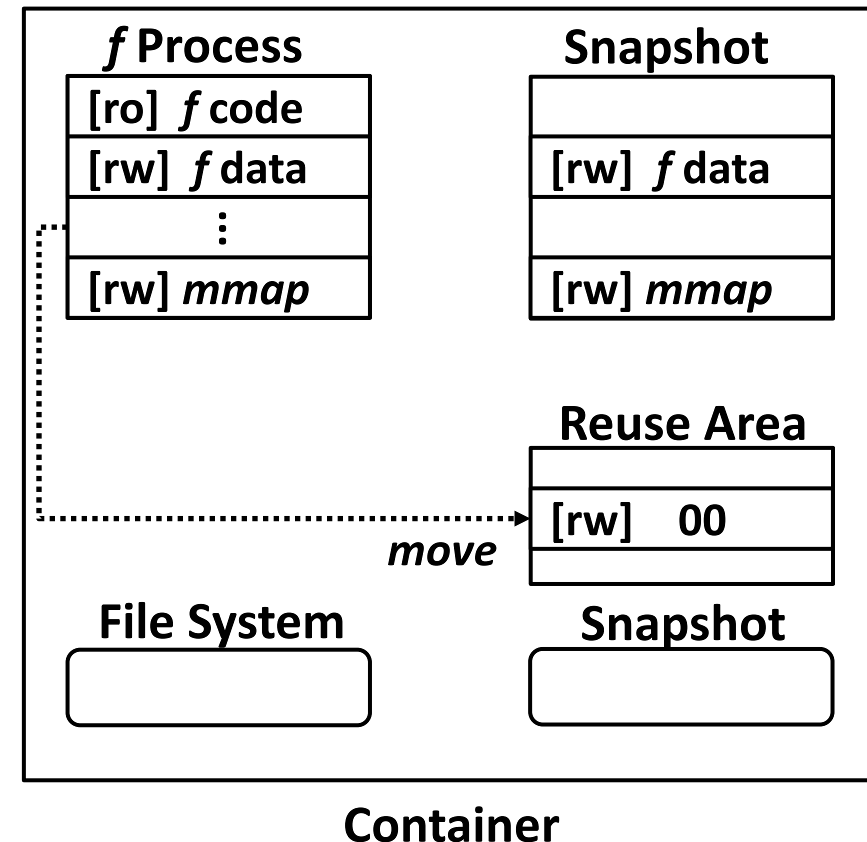
Snapshot/Rewind Operations

- ***rewind***()
  - Restore dirty pages to original ones
    - Reset pages to zero if necessary
  - Delete memory mappings mapped after snapshot
    - Keep pages and related metadata to accelerate *mmap*()s in next function execution



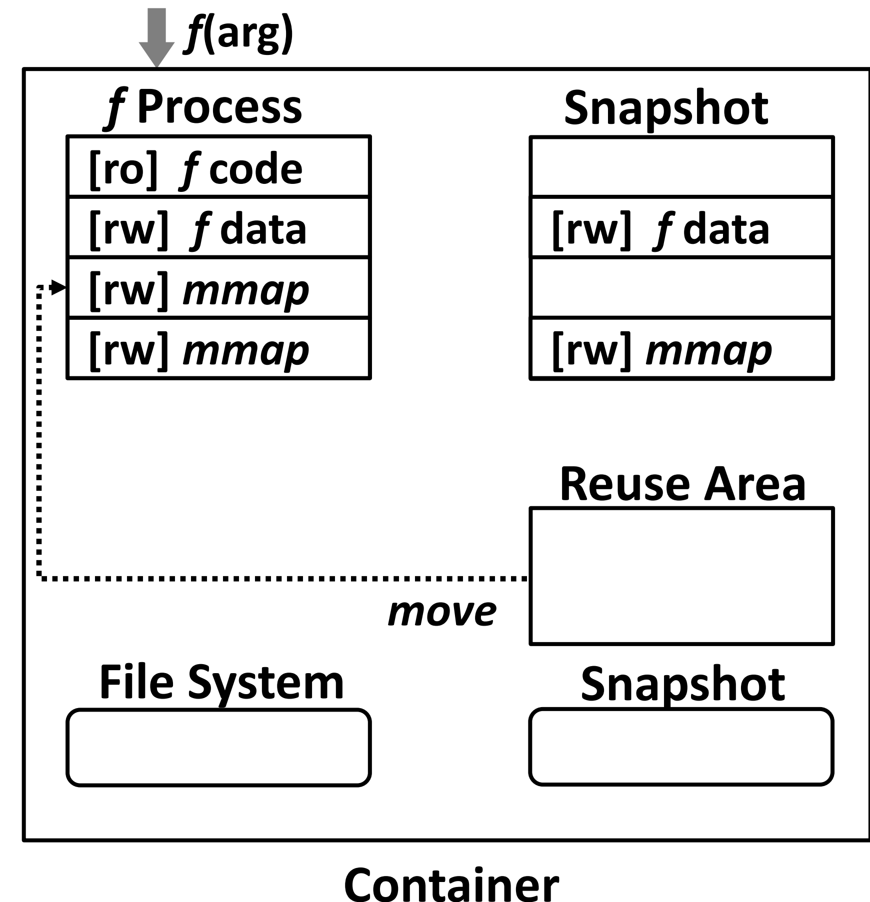
Snapshot/Rewind Operations

- ***rewind***()
  - Restore dirty pages to original ones
    - Reset pages to zero if necessary
  - Delete memory mappings mapped after snapshot
    - Keep pages and related metadata to accelerate *mmap*()s in next function execution



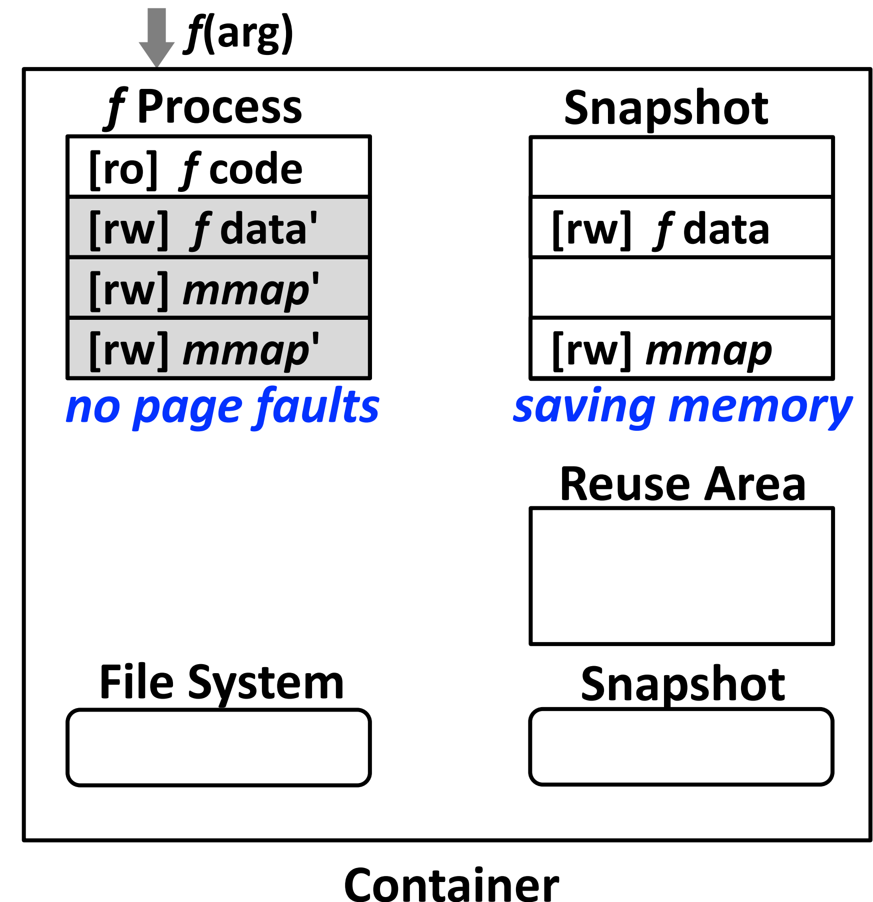
Snapshot/Rewind Operations

- ***rewind***()
  - Restore dirty pages to original ones
    - Reset pages to zero if necessary
  - Delete memory mappings mapped after snapshot
    - Keep pages and related metadata to accelerate *mmap*()s in next function execution



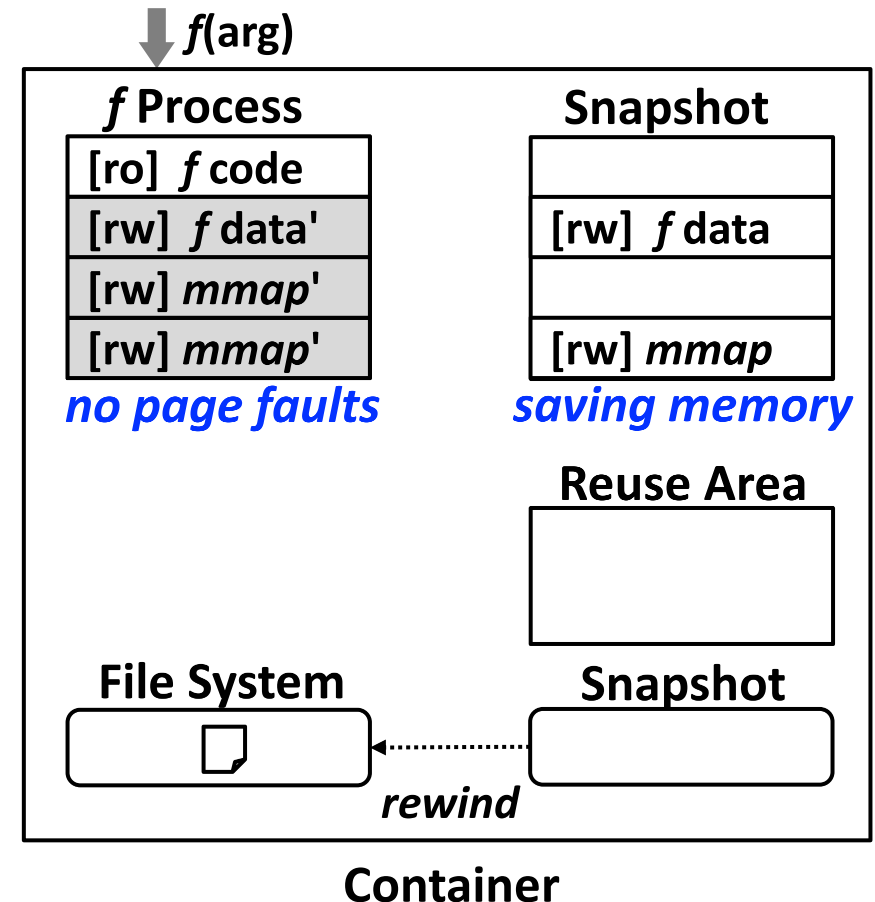
Snapshot/Rewind Operations

- ***rewind*()**
  - Restore dirty pages to original ones
    - Reset pages to zero if necessary
  - Delete memory mappings mapped after snapshot
    - Keep pages and related metadata to accelerate *mmap*()s in next function execution



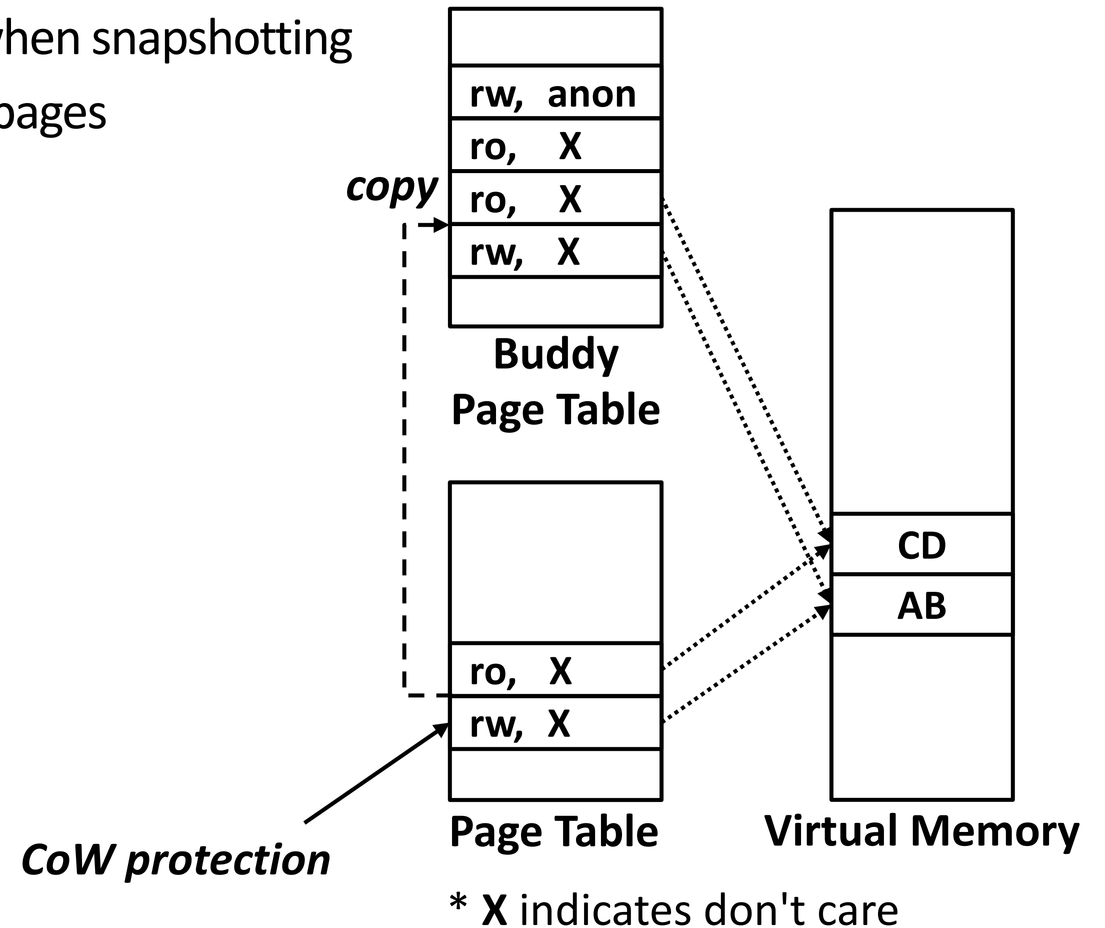
Snapshot/Rewind Operations

- ***rewind***()
  - Restore dirty pages to original ones
    - Reset pages to zero if necessary
  - Delete memory mappings mapped after snapshot
    - Keep pages and related metadata to accelerate *mmap*()s in next function execution
  - Restore files to the snapshot



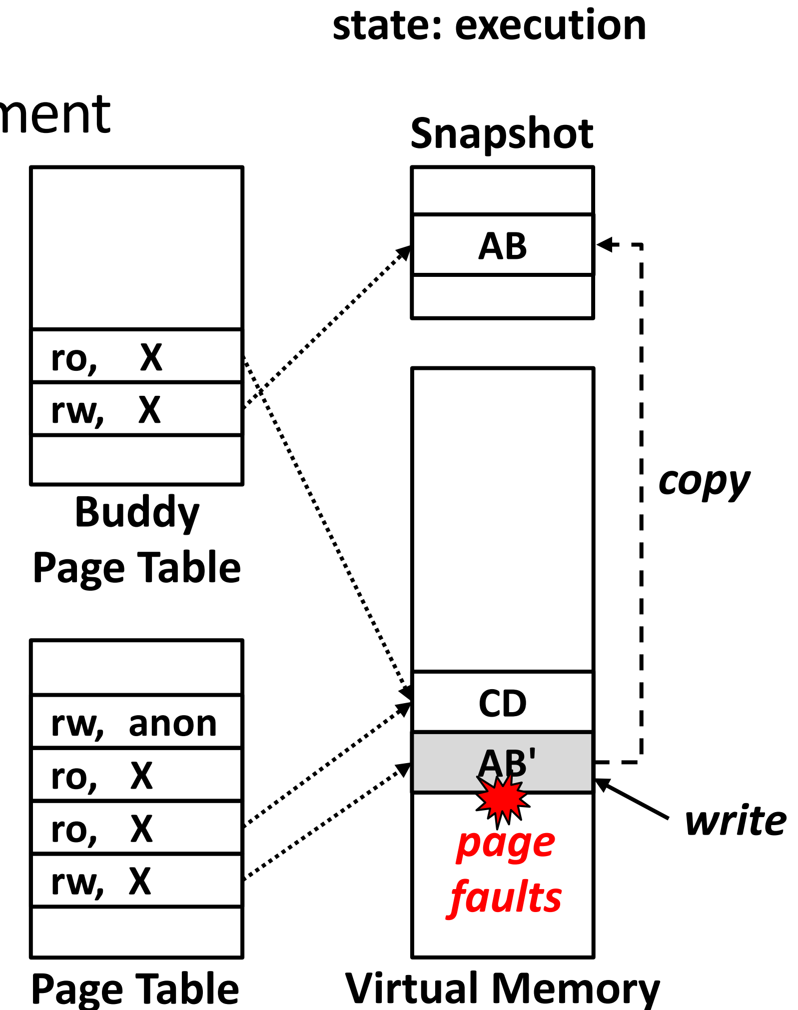
Snapshot/Rewind Operations

- Efficient **kernel-level** snapshot/rewind
  - Introduce *buddy page table* for efficient snapshot management
    - Copy original page table to buddy page table when snapshotting
    - Enable copy-on-write protection to track dirty pages



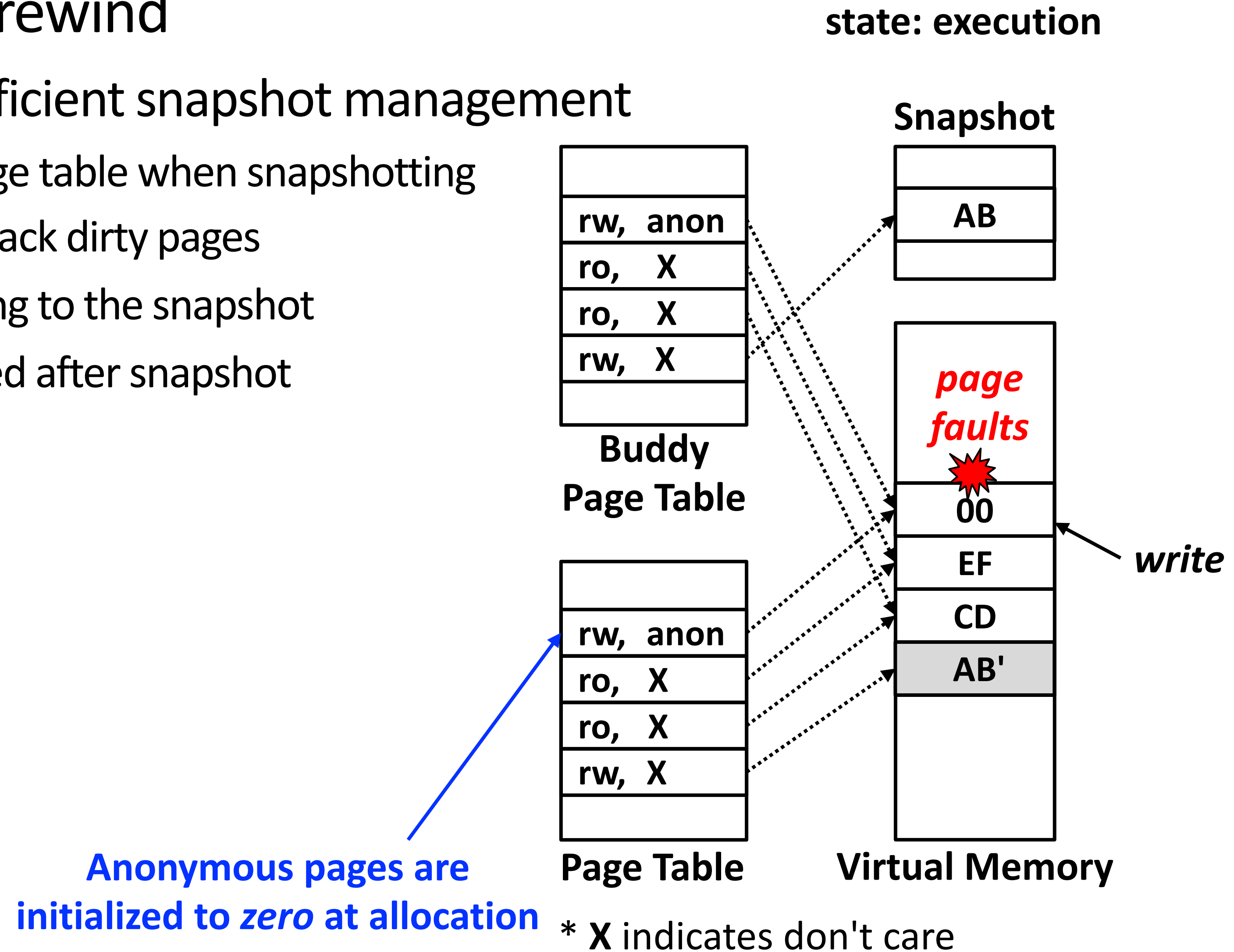
Snapshot/Rewind Operations

- Efficient **kernel-level** snapshot/rewind
  - Introduce *buddy page table* for efficient snapshot management
    - Copy original page table to buddy page table when snapshotting
    - Enable copy-on-write protection to track dirty pages
    - Copy a page during page fault handling to the snapshot



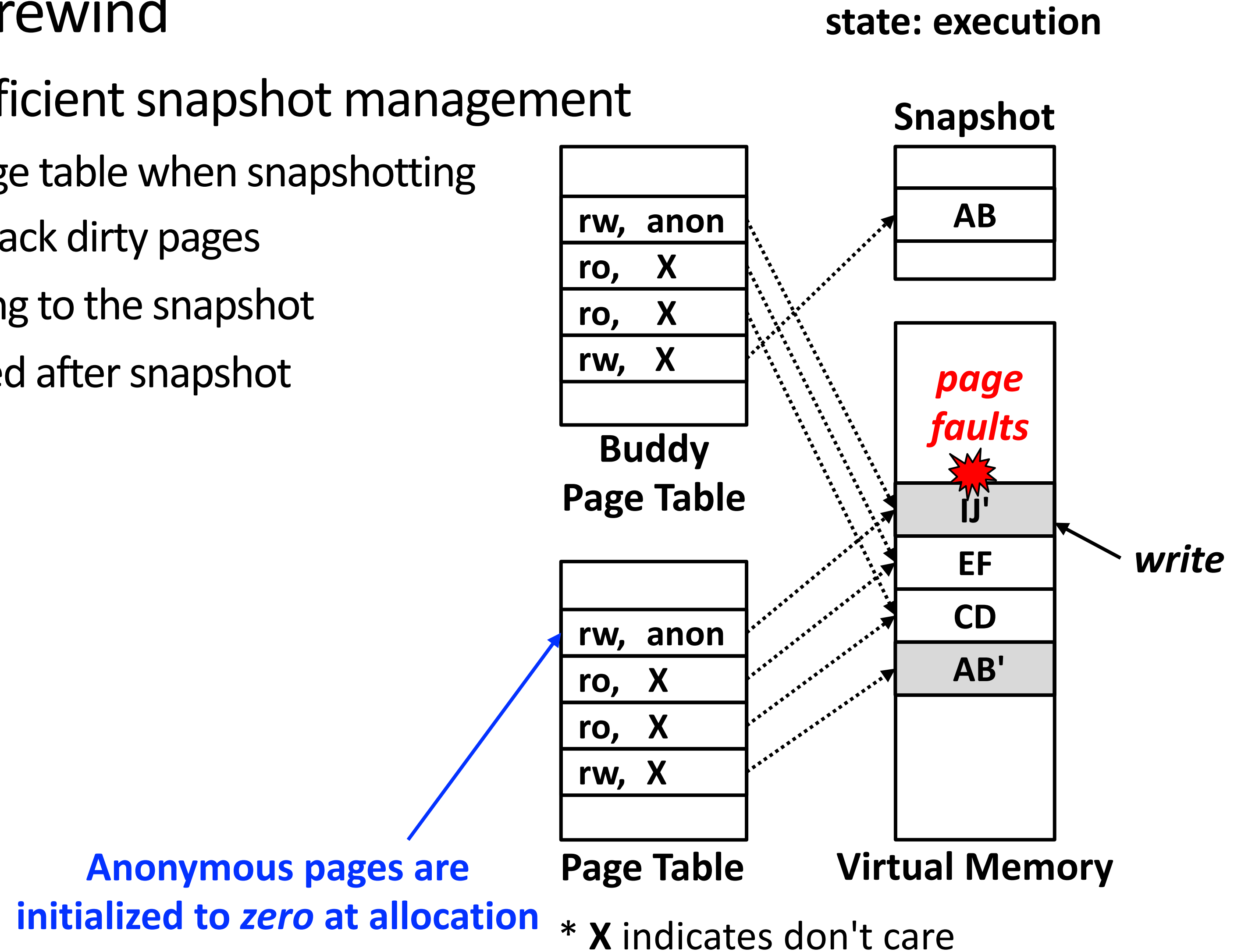
Snapshot/Rewind Operations

- Efficient **kernel-level** snapshot/rewind
  - Introduce *buddy page table* for efficient snapshot management
    - Copy original page table to buddy page table when snapshotting
    - Enable copy-on-write protection to track dirty pages
    - Copy a page during page fault handling to the snapshot
      - Zero (anonymous) pages allocated after snapshot are not copied



Snapshot/Rewind Operations

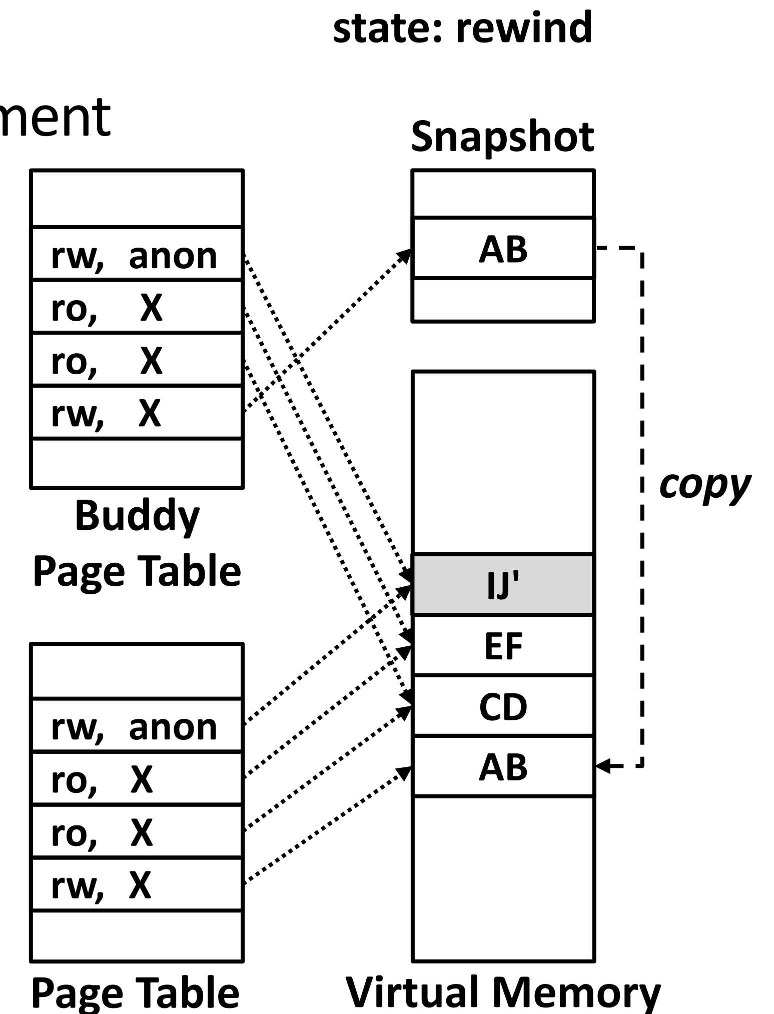
- Efficient **kernel-level** snapshot/rewind
  - Introduce *buddy page table* for efficient snapshot management
    - Copy original page table to buddy page table when snapshotting
    - Enable copy-on-write protection to track dirty pages
    - Copy a page during page fault handling to the snapshot
      - Zero (anonymous) pages allocated after snapshot are not copied



Snapshot/Rewind Operations

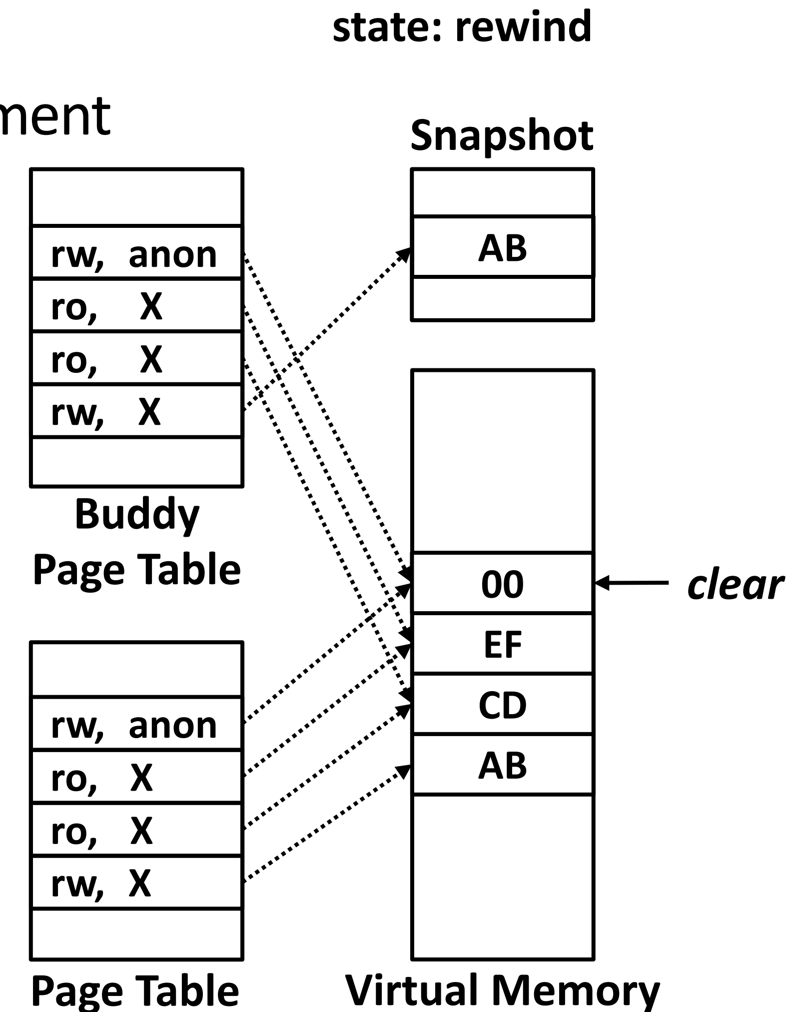
- Efficient **kernel-level** snapshot/rewind
  - Introduce *buddy page table* for efficient snapshot management
    - Copy original page table to buddy page table when snapshotting
    - Enable copy-on-write protection to track dirty pages
    - Copy a page during page fault handling to the snapshot
      - Zero (anonymous) pages allocated after snapshot are not copied
  - Rewind dirty pages to the snapshot
    - Copy back snapshot pages to restore to initial state
      - Allow write permission

      → No page faults on repeated execution



Snapshot/Rewind Operations

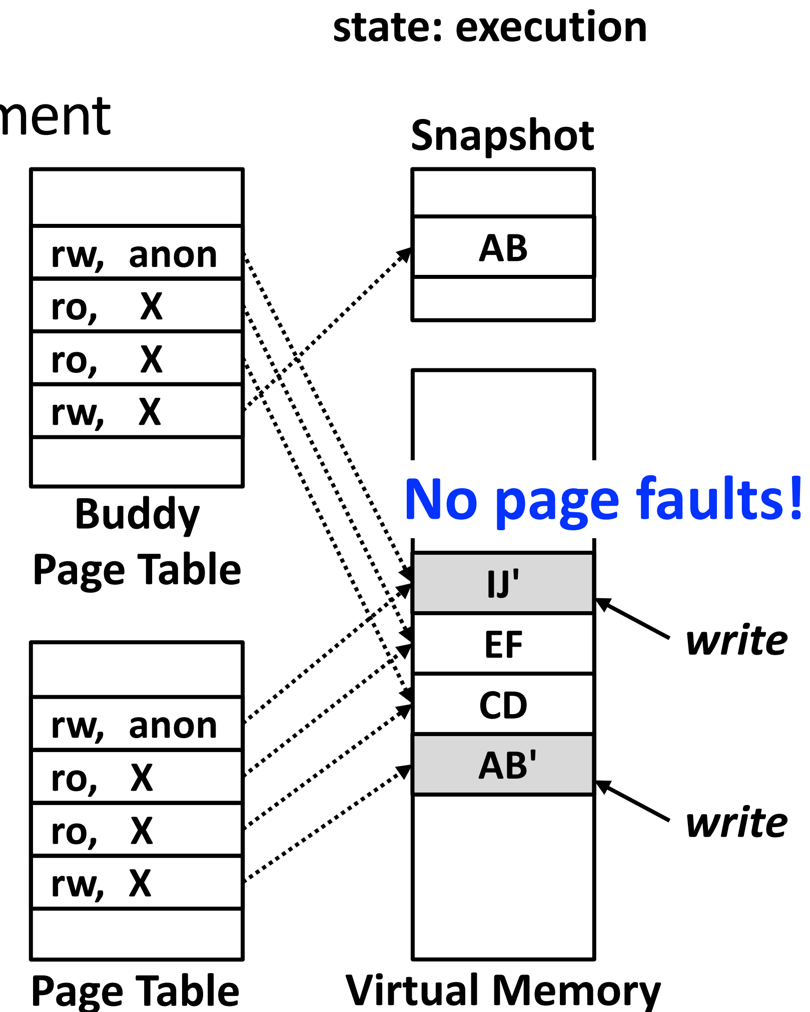
- Efficient **kernel-level** snapshot/rewind
  - Introduce *buddy page table* for efficient snapshot management
    - Copy original page table to buddy page table when snapshotting
    - Enable copy-on-write protection to track dirty pages
    - Copy a page during page fault handling to the snapshot
      - Zero (anonymous) pages allocated after snapshot are not copied
  - Rewind dirty pages to the snapshot
    - Copy back snapshot pages to restore to initial state
      - Allow write permission
      - → No page faults on repeated execution
    - Clear anonymous pages allocated after snapshot to zero



Snapshot/Rewind Operations

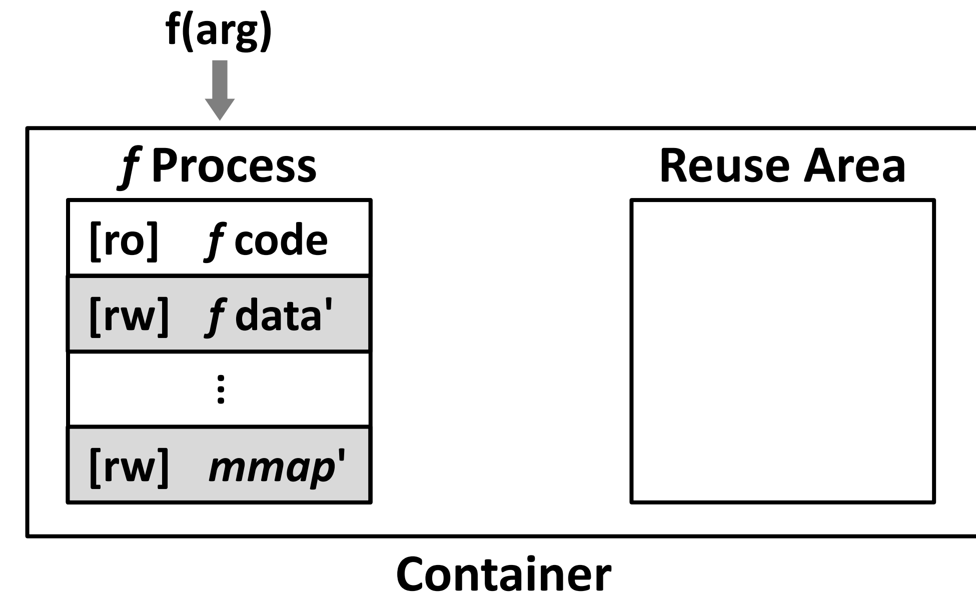
- Efficient **kernel-level** snapshot/rewind
  - Introduce *buddy page table* for efficient snapshot management
    - Copy original page table to buddy page table when snapshotting
    - Enable copy-on-write protection to track dirty pages
    - Copy a page during page fault handling to the snapshot
      - Zero (anonymous) pages allocated after snapshot are not copied
  - Rewind dirty pages to the snapshot
    - Copy back snapshot pages to restore to initial state
      - Allow write permission

      → No page faults on repeated execution
    - Clear anonymous pages allocated after snapshot to zero



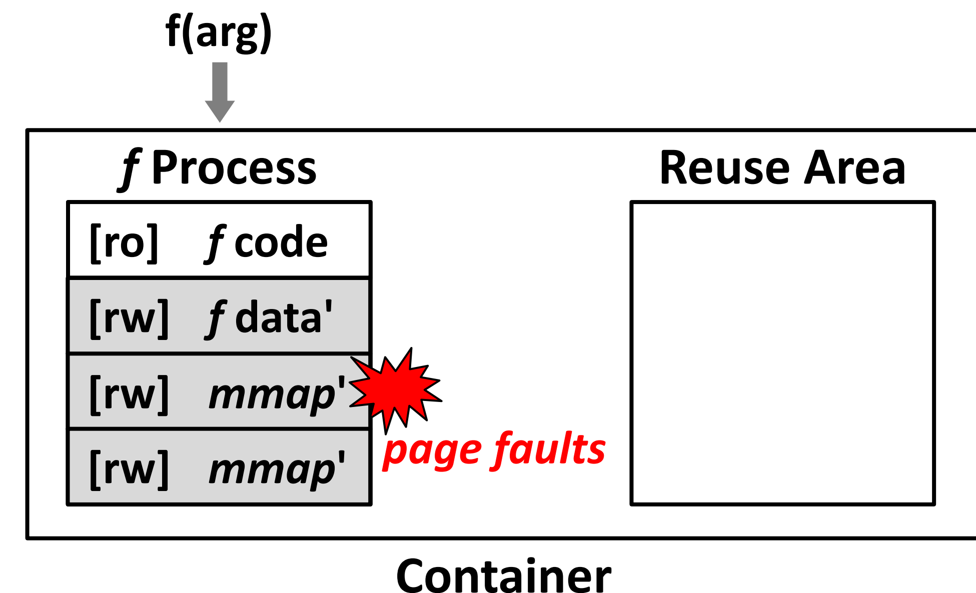
Memory Area Reuse

- Additional memory area can be allocated and used after snapshot
  - *mmap*() followed by page fault and page allocation
  - These overheads repeat on every function request handling



Memory Area Reuse

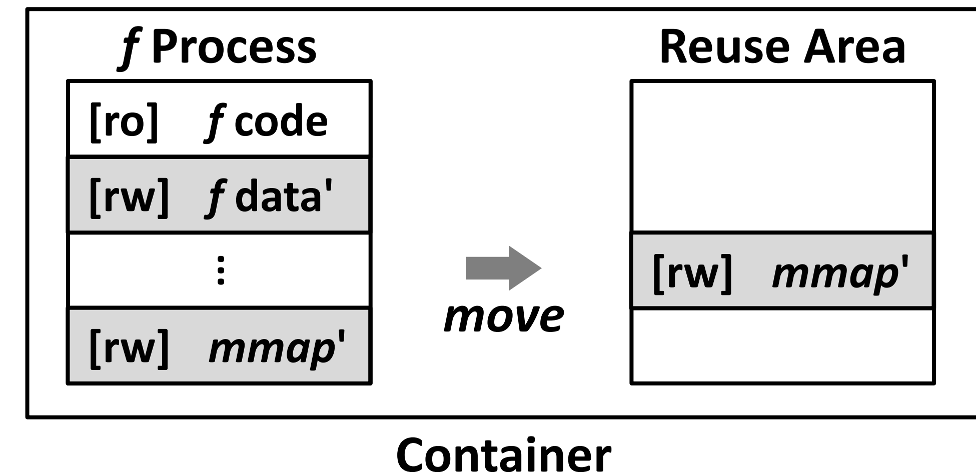
- Additional memory area can be allocated and used after snapshot
  - *mmap*() followed by page fault and page allocation
  - These overheads repeat on every function request handling
- Memory area reuse minimizes overhead from page faults of new *mmaps*



Memory Area Reuse

- Additional memory area can be allocated and used after snapshot
  - *mmap*() followed by page fault and page allocation
  - These overheads repeat on every function request handling

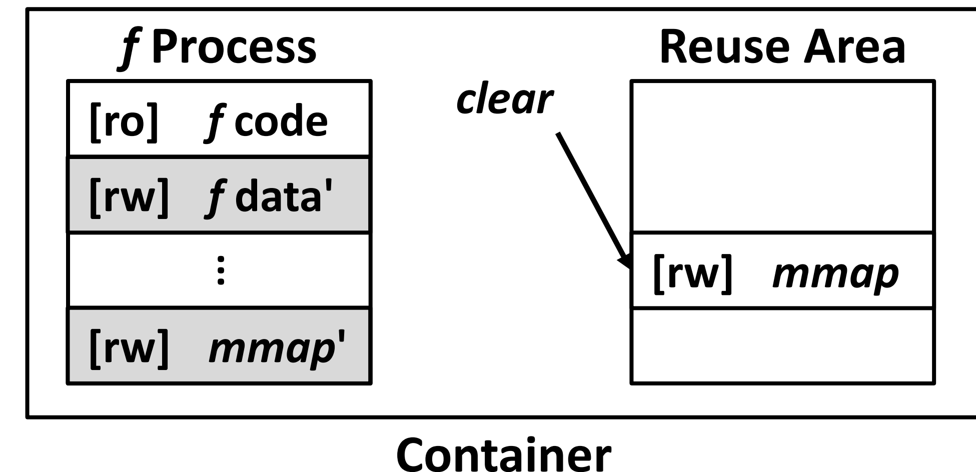
- Memory area reuse minimizes overhead from page faults of new *mmaps*
  - Pages, page tables and associated metadata are reused in the next function execution



Memory Area Reuse

- Additional memory area can be allocated and used after snapshot
  - *mmap*() followed by page fault and page allocation
  - These overheads repeat on every function request handling

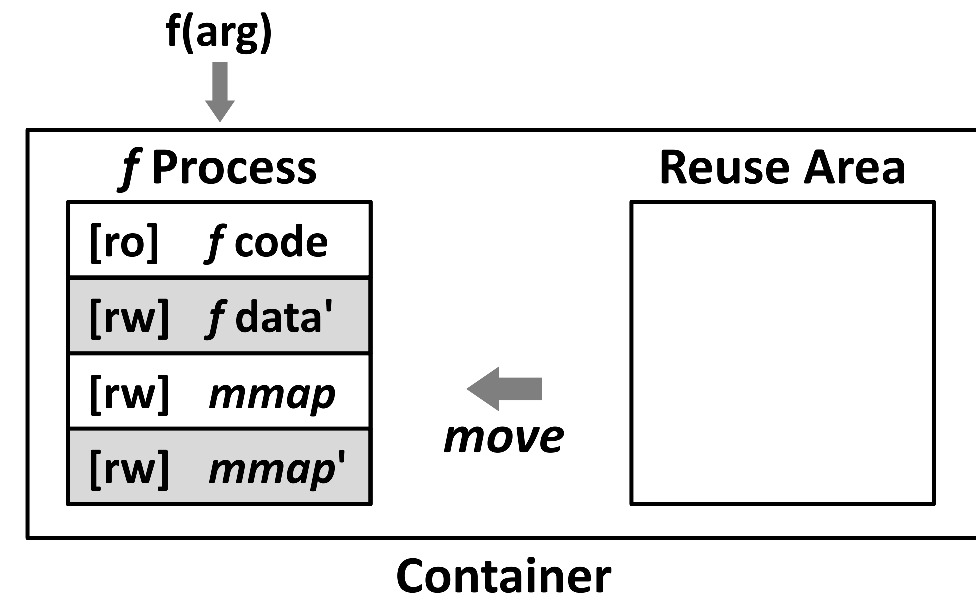
- Memory area reuse minimizes overhead from page faults of new *mmaps*
  - Pages, page tables and associated metadata are reused in the next function execution
  - Pages are cleared to zero to prevent data leakage



Memory Area Reuse

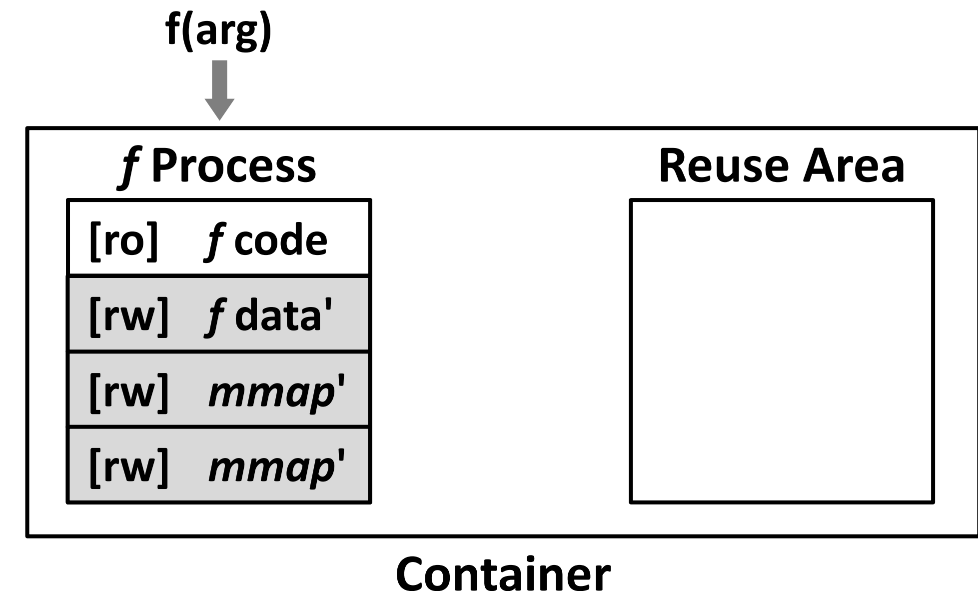
- Additional memory area can be allocated and used after snapshot
  - *mmap*() followed by page fault and page allocation
  - These overheads repeat on every function request handling

- Memory area reuse minimizes overhead from page faults of new *mmaps*
  - Pages, page tables and associated metadata are reused in the next function execution
  - Pages are cleared to zero to prevent data leakage



Memory Area Reuse

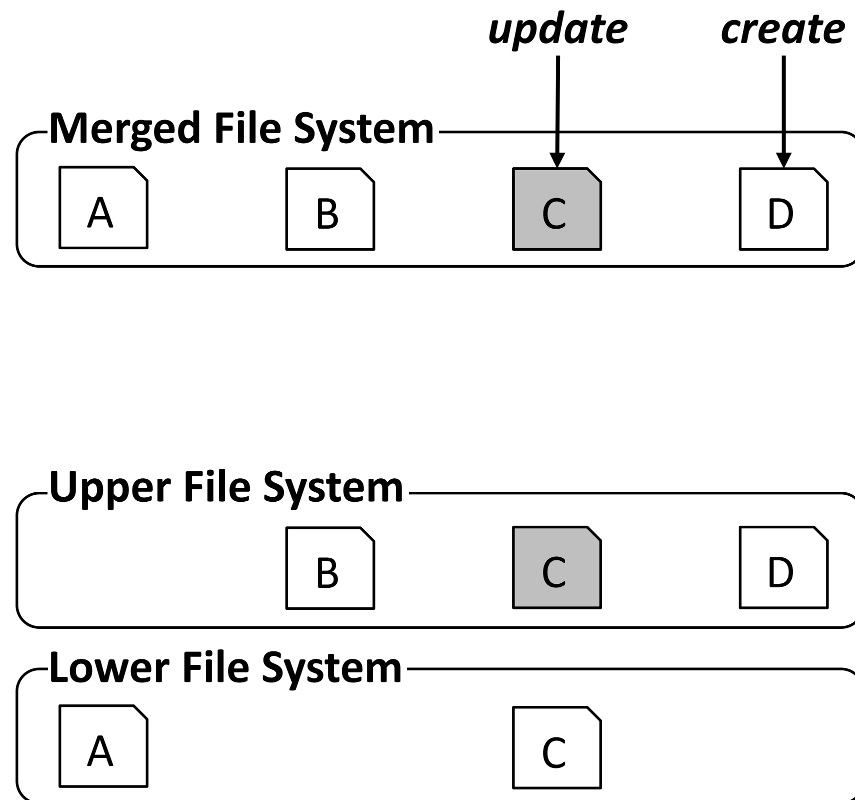
- Additional memory area can be allocated and used after snapshot
  - *mmap*() followed by page fault and page allocation
  - These overheads repeat on every function request handling
- Memory area reuse minimizes overhead from page faults of new *mmaps*
  - Pages, page tables and associated metadata are reused in the next function execution
  - Pages are cleared to zero to prevent data leakage
  - Reuse is limited to anonymous memory

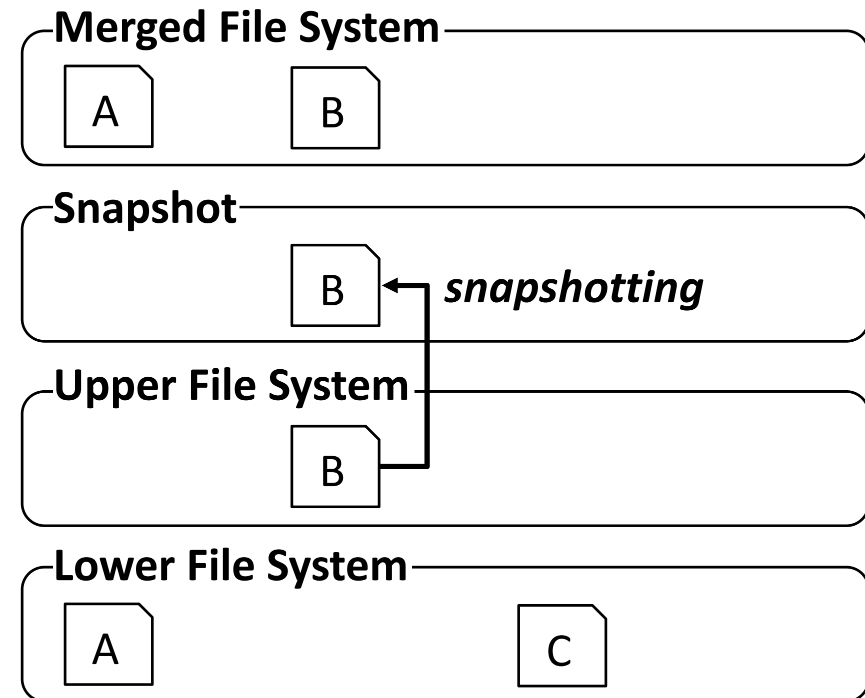


Good performance: no page faults!

Remove File Persistence

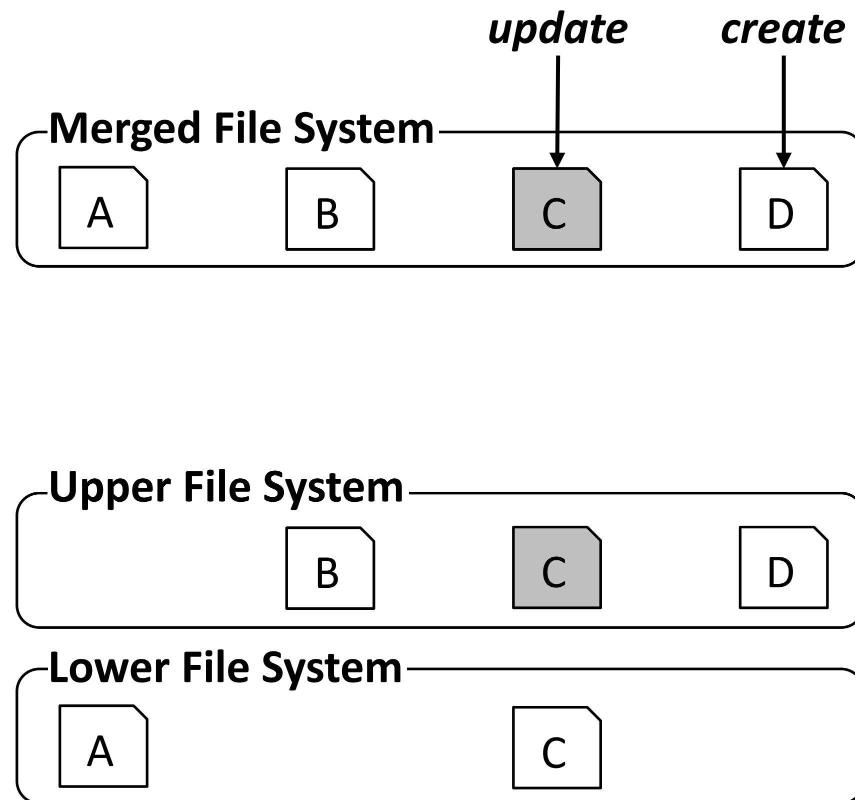
- File persistence is removed by rewinding the file system from the snapshot
  - User-level implementation on OverlayFS (file system used by Docker)

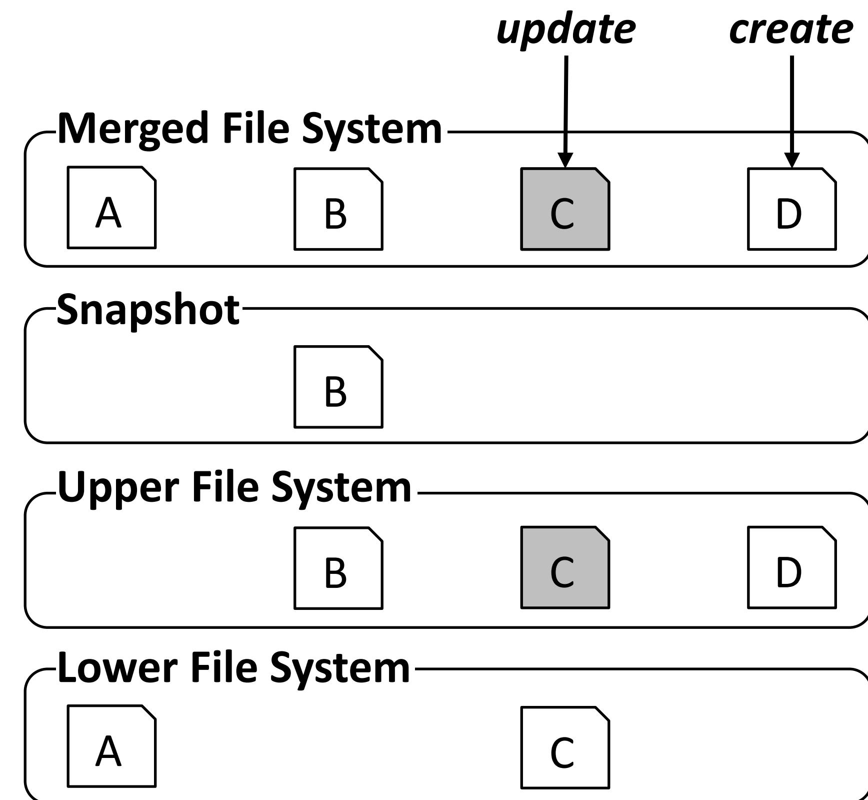




Remove File Persistence

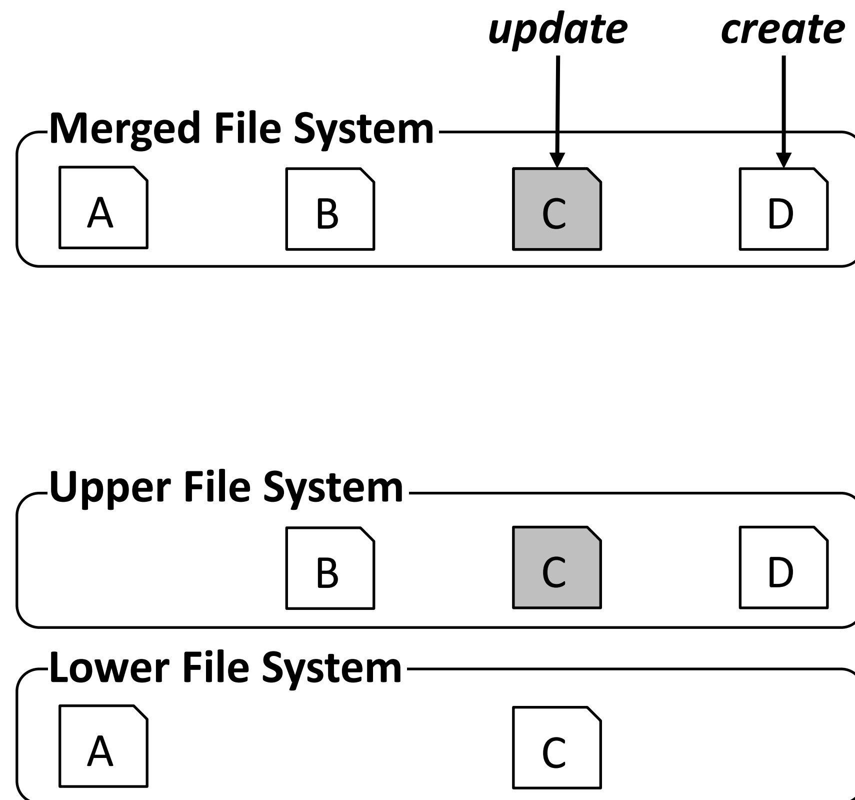
- File persistence is removed by rewinding the file system from the snapshot
  - User-level implementation on OverlayFS (file system used by Docker)

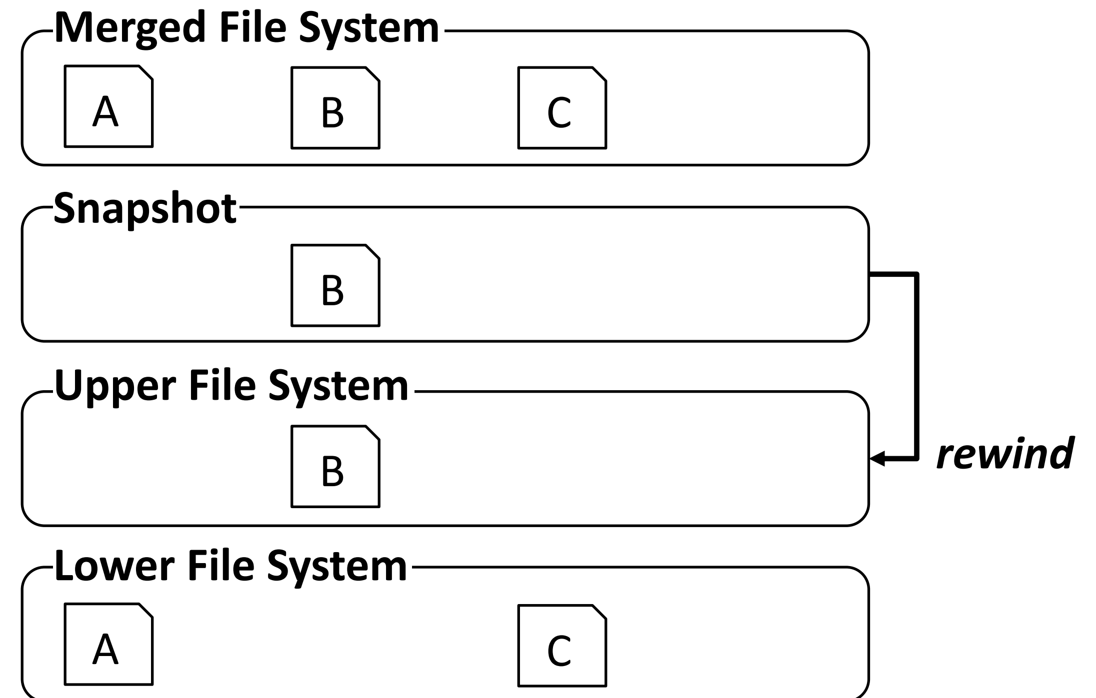




Remove File Persistence

- File persistence is removed by rewinding the file system from the snapshot
  - User-level implementation on OverlayFS (file system used by Docker)





Evaluation

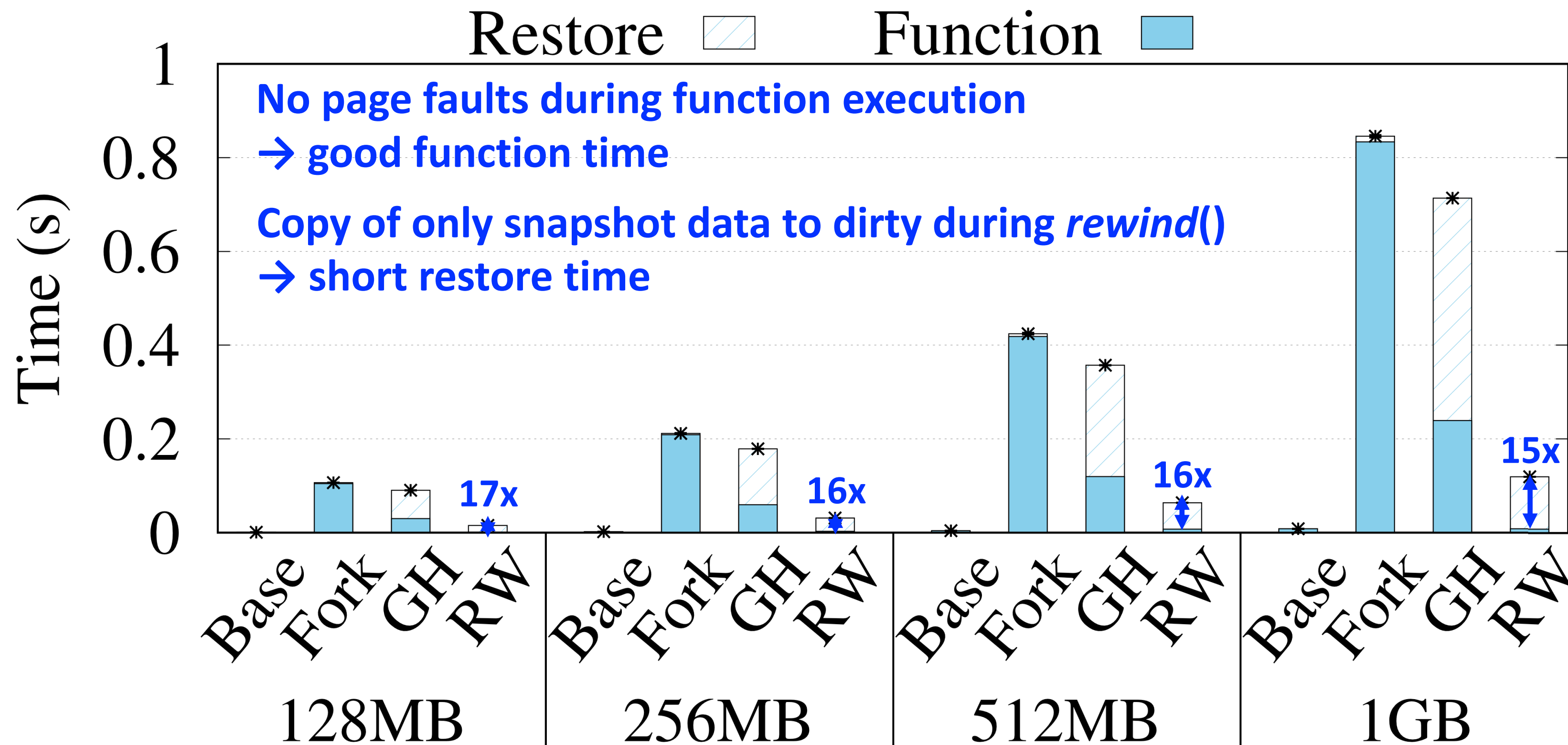
- Key questions:
  - Does REWIND effectively save memory size of the snapshot?
  - How much do the snapshot/rewind operations impact function execution time?
    - How much does REWIND accelerate function execution time?

- Comparison with
  - Baseline – execute function with container reuse
  - Fork – employ the *fork*() system call on the baseline
  - Groundhog (GH) – create a snapshot of a function process and restore to the snapshot

[1] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

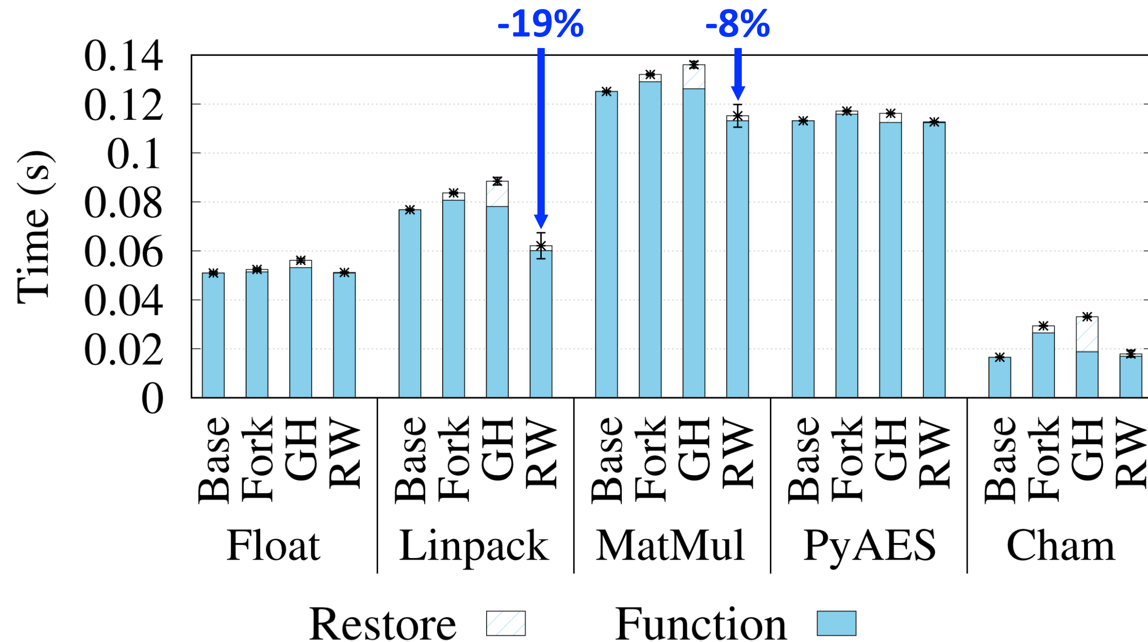
Microbenchmark

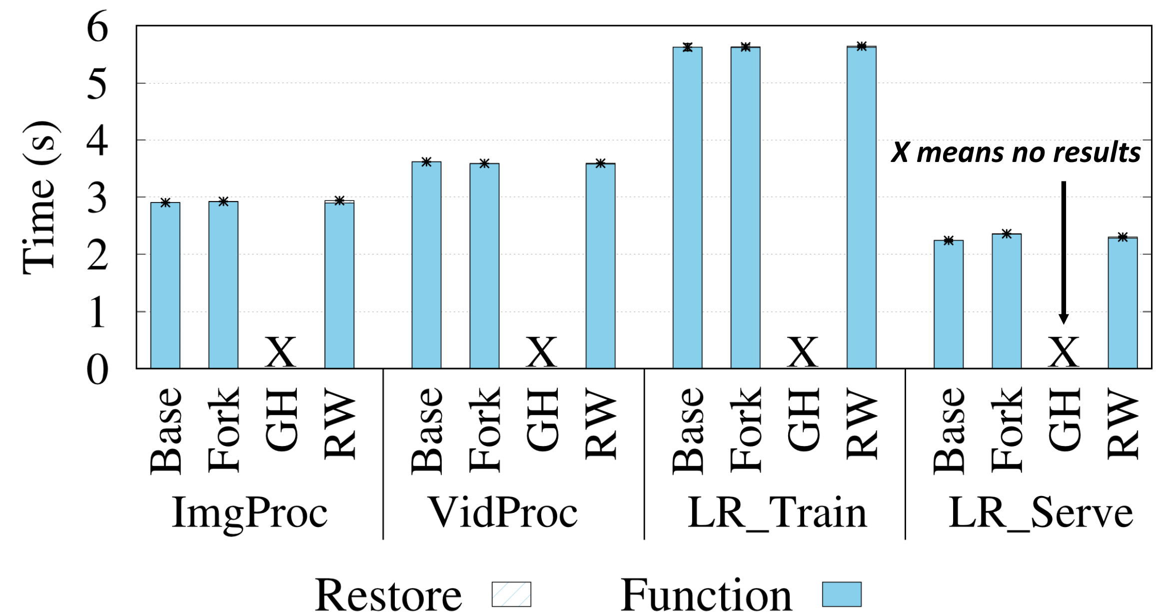
- REWIND shows less overheads than Fork and GH
  - 1:1 ratio of random read/write
  - Increase memory working set size 128MB to 1GB



Function Latency

- REWIND shows better performance even than the baseline
  - Real workloads – FunctionBench [1]
  - Break down the latency into function time and restore time

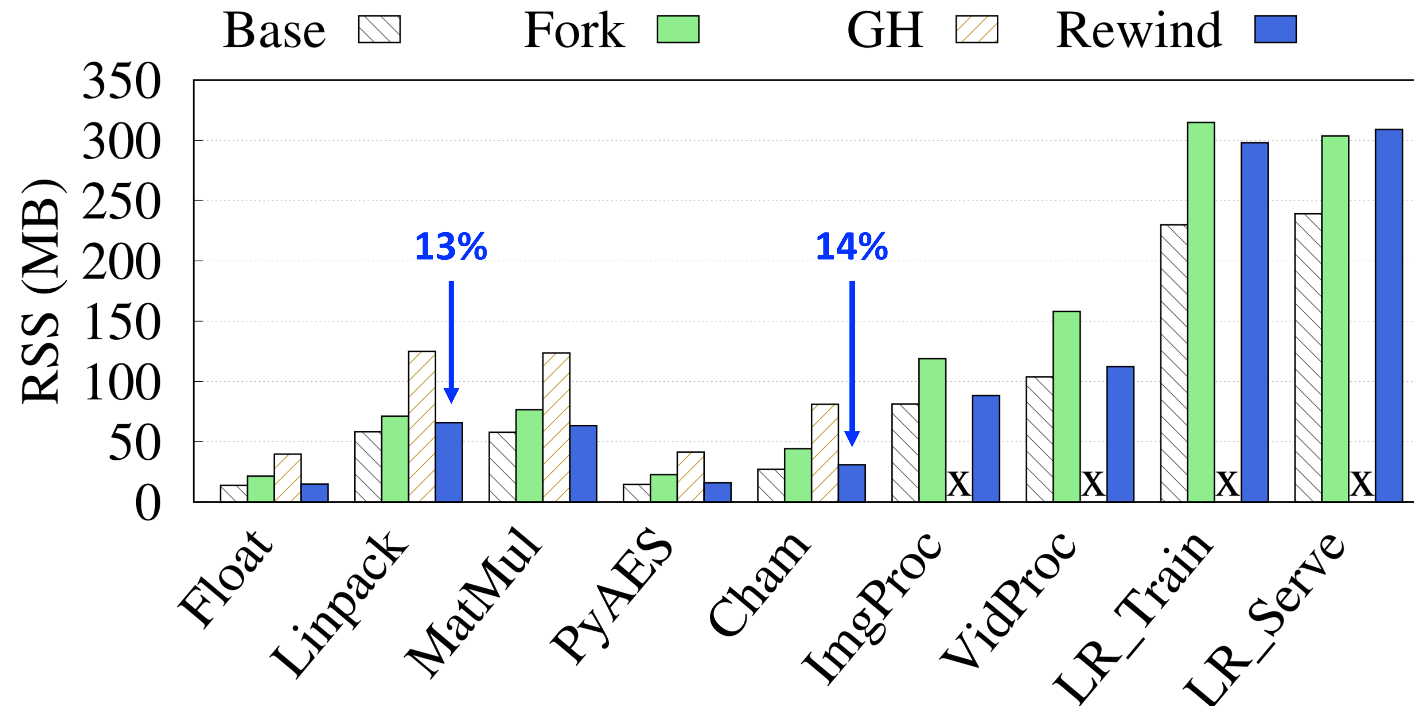




[1] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

Memory Consumption

- REWIND consumes lower memory than Fork and GH
  - Real workloads – FunctionBench
  - Measure peak memory usage (RSS)



REWIND makes a copy of only dirty data in the snapshot → Low memory usage

Conclusion

- REWIND: secure, fast, and resource-efficient serverless platform
  - Security: remove **quasi-persistence** of data in containers
  - Performance: provide efficient ***snapshot***/***rewind*** and **reuse memory** for next run
  - Resource usage: do **not copy all data** to the snapshot

REWIND is available at:
https://github.com/s3yonsei/rewind_serverless

Thank you!

Backup