Monitoring Access to Shared Memory-Mapped Files

By

Christian Sarmoria, Steve Chapin

Presented At

The Digital Forensic Research Conference

DFRWS 2005 USA New Orleans, LA (Aug 17th - 19th)

Monitoring Access to Shared Memory-Mapped Files

Christian G. Sarmoria, Steve J. Chapin
{cgsarmor , chapin} @ecs.syr.edu

Electrical Engineering and Computer Science Dept.
Syracuse University
NY, 13210

August 18, 2005

Overview

- Introduction
- Reconstruction of sequences of events
- Shared memory
- Granularity
- Page-level monitor
- Conclusions
- Q & A

Evidence Gathering

- ***Post-mortem***
  - Filesystem
  - Memory dump
  - Available logs

- **Realtime**
  - System, application, and network logs
  - ***Record system objects and their interactions***
  - <u>Evidence is *potential*</u>

Framework for Reconstructing Sequences of Events

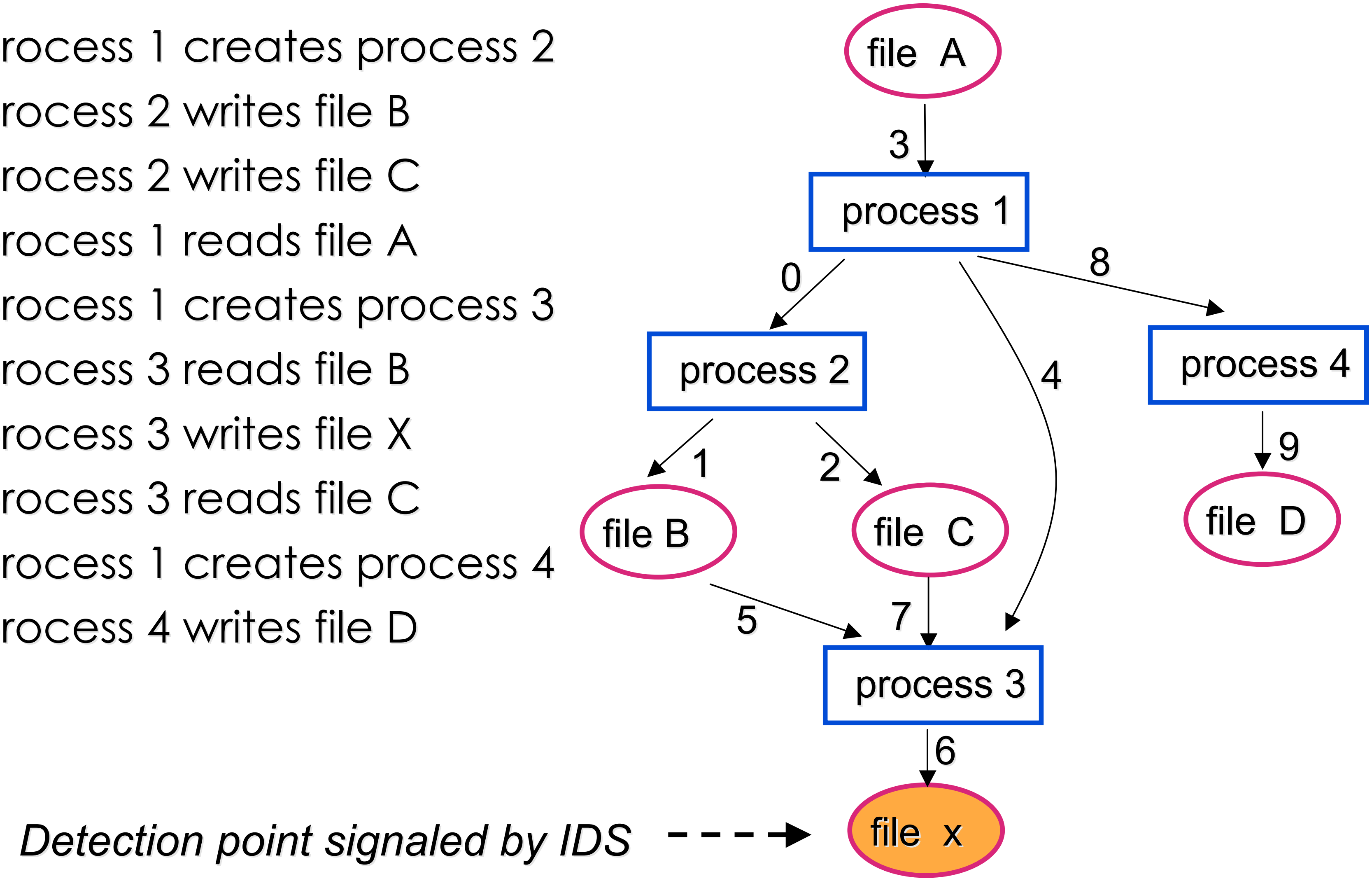
- Monitor system **objects** and **events**
- Gather potential evidence at runtime
- Detection point signaled by an IDS
- Build dependency graph
- Backtrack from detection point back to entry point of the attack

Framework for Reconstructing Sequences of Events

- **System objects:** Process, file, and filename.
- **Event**: Read, write, rename, create process, share memory, etc.
- **Role** of an object in an event: ***cause*** or ***effect***
- **Objects' dependencies**:
  - *Process-Process:* create, signal, share memory
  - *Process-file:* read, write, change attributes
  - *Process-filename:* rename, link, remove, etc.

Framework for Reconstructing Sequences of Events

T_0: process 1 creates process 2
T_1: process 2 writes file B
T_2: process 2 writes file C
T_3: process 1 reads file A
T_4: process 1 creates process 3
T_5: process 3 reads file B
T_6: process 3 writes file X
T_7: process 3 reads file C
T_8: process 1 creates process 4
T_9: process 4 writes file D



Reconstruction of Sequences of Events

- **Current Systems**
  - SNARE (2003)
  - Backtracker (2003)
  - Forensix (2003)
  - CIDS (2004)
  - Improved Backtracker (2005)
- **System calls monitoring**
- **Shared memory?**
  - System call event only (`mmap/munmap`)
  - Aggregation and coarse granularity
  - False dependencies
  - Overlooked events

File access

- Current systems: `read` and `write` system calls
- Works for access to the file in the file system
- Once a file is memory-mapped we lose system calls support
- Process accesses a memory-mapped file through direct address space manipulation (pointers)
- We need to monitor memory *read* and *write* instructions

Shared-Memory Object (SMO)

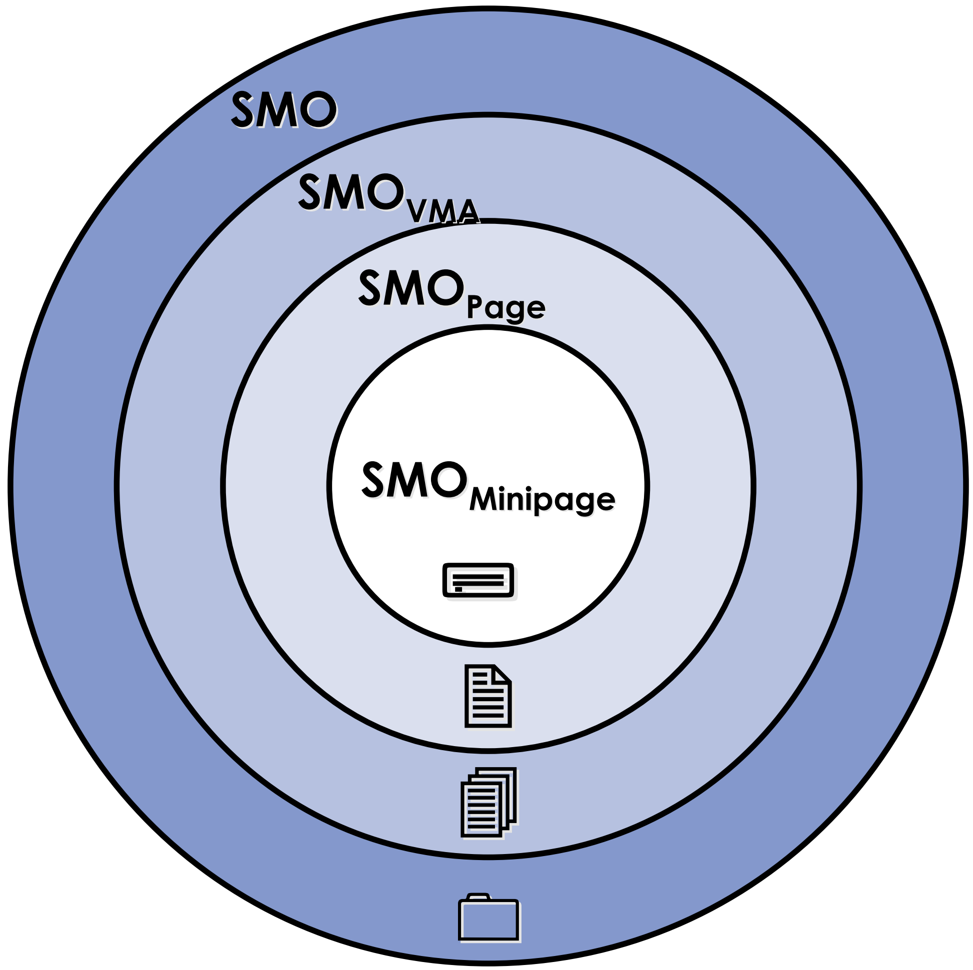
- An *SMO* is a portion of memory address space allocated to a process and shared with other processes
- Roles:
  - *Cause*
  - *Effect*
- Process-SMO dependencies:
  - Process ⇒ SMO
  - SMO ⇒ Process

Granularity

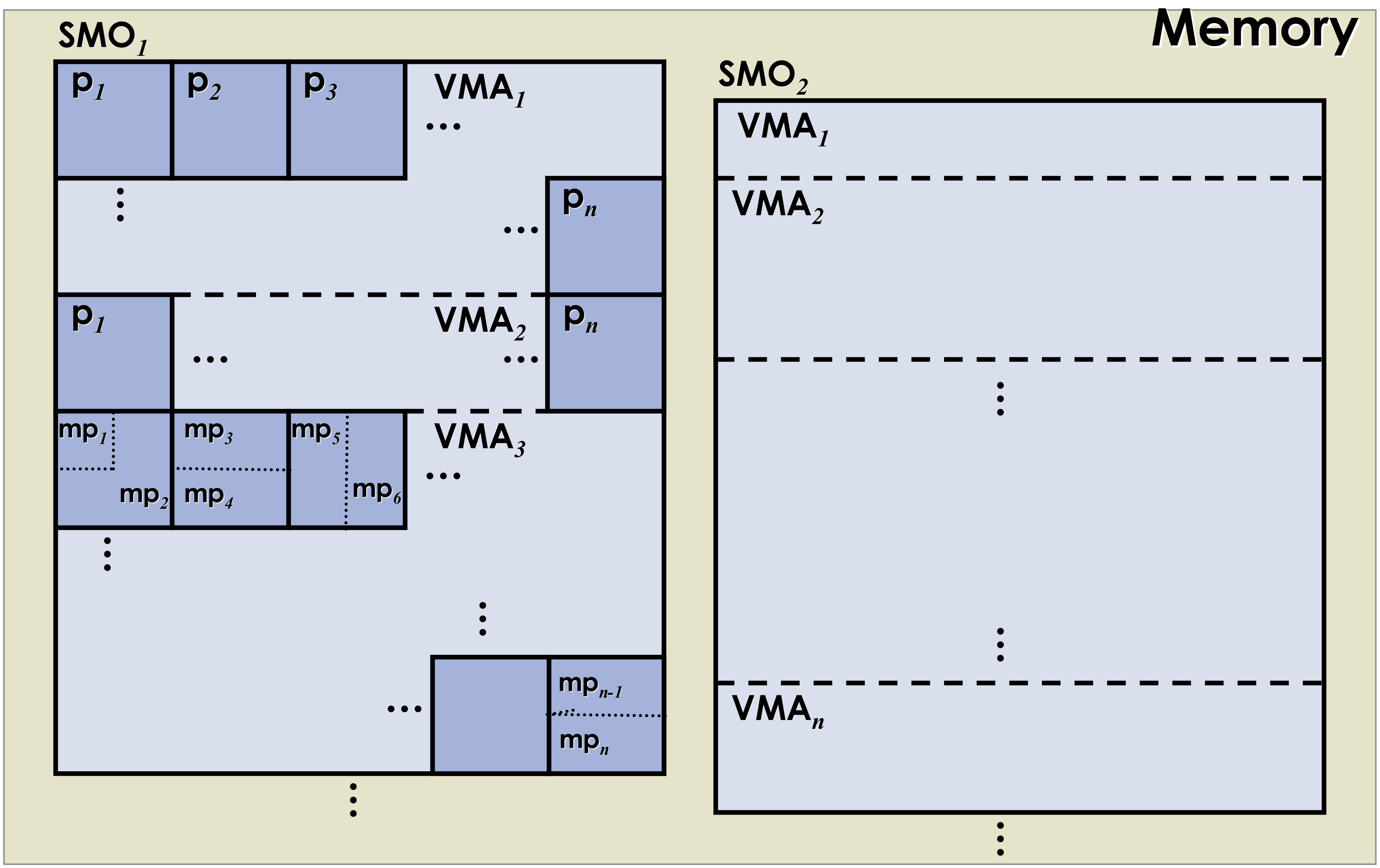
- SMO composed by constituent parts
- A *part* is a section of the address space occupied by the SMO

Constituent Parts	Size
Object	The whole SMO
Memory Region (VMA)	Runtime-OS-Defined set of contiguous memory frames
Page	As defined in the OS, usually 4KB
Minipage	1 byte ≤ *minipage* < page

Granularity



Granularity



Granularity: Properties

- Monitoring constituent parts eliminates a number of Process-Process false dependencies
- The larger the SMO the greater the benefit of applying granularity
- Monitoring at the *minipage* level has to deal with loss of paging hardware support
- Overhead

Technique

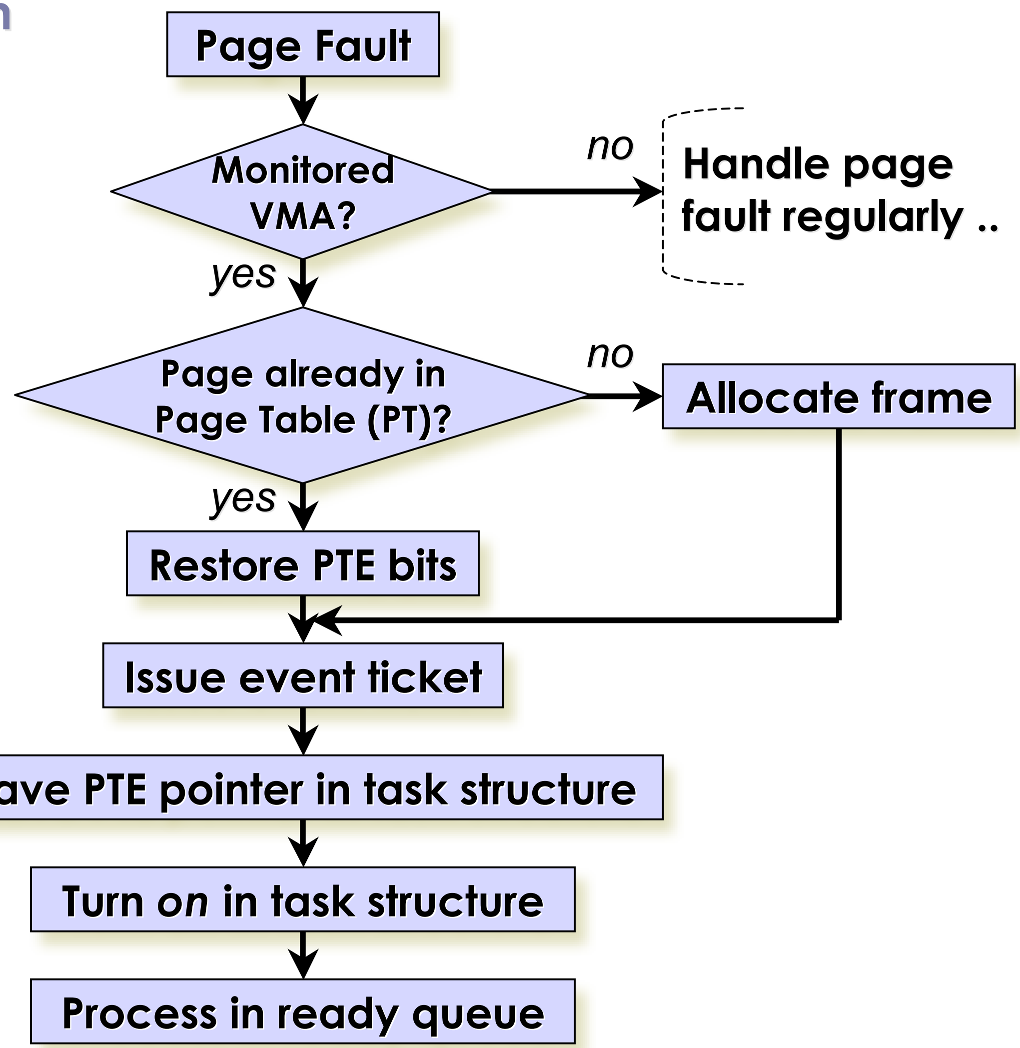
- Linux Kernel 2.4 on x86 architecture
- Does not require processes' source code
- Accesses to shared memory-mapped files at the page level of granularity
- Runtime SMO registration by intercepting `mmap` system call
- Monitor implemented in page fault handler and CPU scheduler
- Event logs saved in circular buffer in kernel space

Technique

- Memory pages where the SMO resides are set protected and read-only through Page Table Entry (PTE)
- Page fault alerts the monitor of SMO access event
- Race condition:
  - Time of page serviced and ready to use is no guarantee of read/write event at present time
  - CPU scheduler confirms event is effective

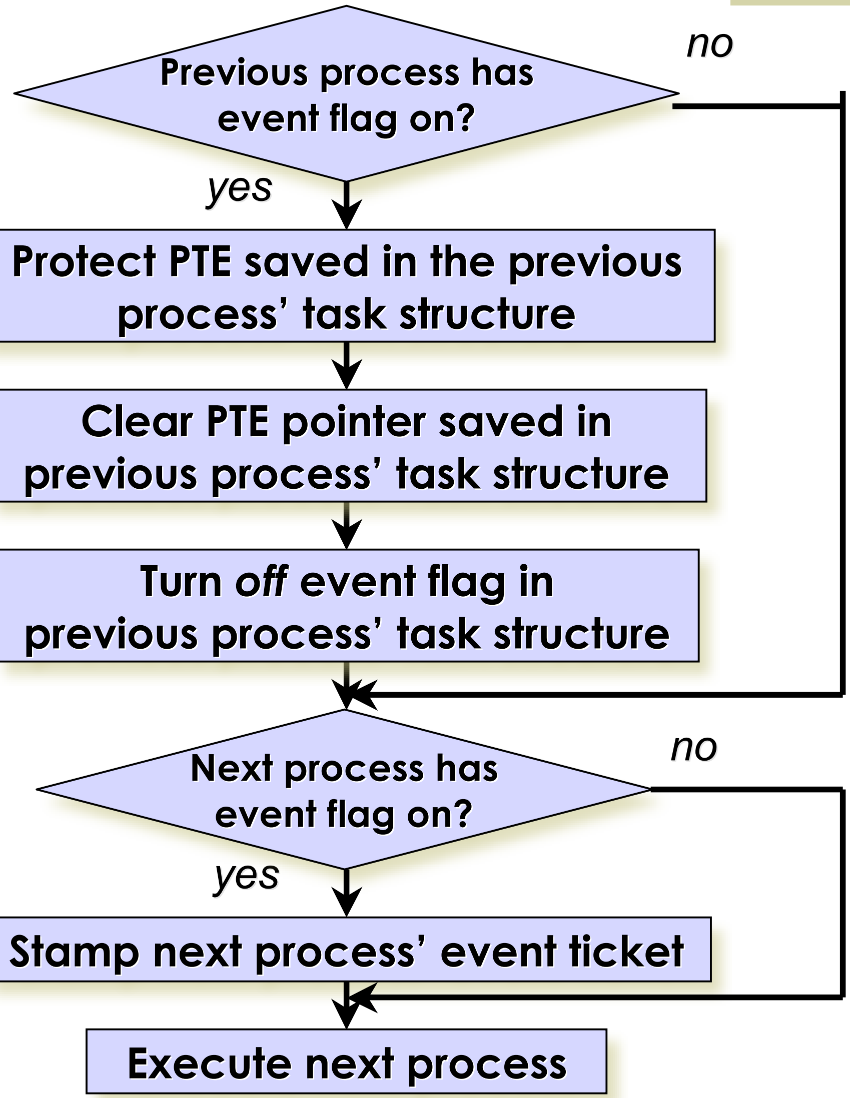
Page Fault Handler

Page Fault Handler with
Page-Level Monitoring



CPU Scheduler

CPU Scheduler with Page-Level Monitoring



Synchronization

- **Race condition:**
  - P_1 and P_2 share a page
  - P_1 page faults on it
  - Page fault handler serves the page for P_1
  - Monitor logs the event
  - P_1 ready for execution, waiting for CPU
  - P_2 page faults on same page
  - Page fault handler serves the page for P_2
  - Monitor logs event
  - P_2 ready for execution
  - P_2 gets CPU before P_1

Results

- Established Process-SMO dependencies for shared memory-mapped files
- x86 limitation: a read after a write in the same page during the same CPU time slice

Results

- Overhead:
  - **One page fault** for any number of **consecutive reads** in the same page during the same CPU time slice
  - **One page fault** for any number of **consecutive writes** in the same page during the same CPU time slice
  - **Two page faults** for any number of **consecutive reads followed** by any number **of consecutive writes** in the same page during the same CPU time slice

Conclusions

- Shared memory is a source of objects of different types which participate in events the same way as other objects do
- OS system call level limitation to monitor shared memory produces false dependencies and overlooked events in current reconstruction systems
- Monitoring SMOs can eliminate a number of false dependencies and reveal others
- Granularity provides a framework to trade off between accuracy and overhead

Questions…