Deadlock

Don Porter

Portions courtesy Emmett Witchel

Concurrency Issues

- Past lectures:
  - Problem: Safely coordinate access to shared resource
  - Solutions:
    - Use semaphores, monitors, locks, condition variables
    - Coordinate access *within* shared objects
- What about coordinated access *across* multiple objects?
  - If you are not careful, it can lead to *deadlock*
- Today's lecture:
  - What is deadlock?
  - How can we address deadlock?

Deadlock: Motivating Examples

- Two *producer* processes share a buffer but use a different protocol for accessing the buffers

```
Producer1() {
  Lock(emptyBuffer)
  Lock(producerMutexLock)
   :
}
```

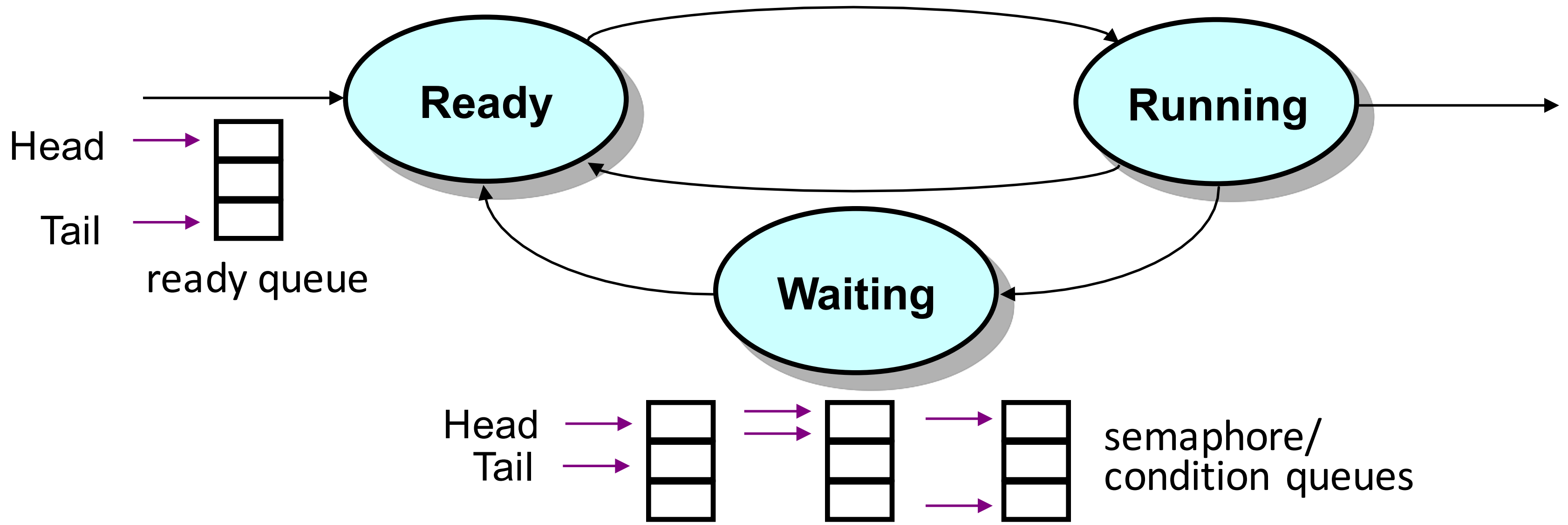
```
Producer2(){
  Lock(producerMutexLock)
  Lock(emptyBuffer)
   :
}
```

- A postscript interpreter and a visualization program compete for memory frames

```
PS_Interpreter(){
  request(memory_frames, 10)
  <process file>
  request(frame_buffer, 1)
  <draw file on screen>
}
```

```
Visualize() {
  request(frame_buffer, 1)
  <display data>
  request(memory_frames, 20)
  <update display>
}
```

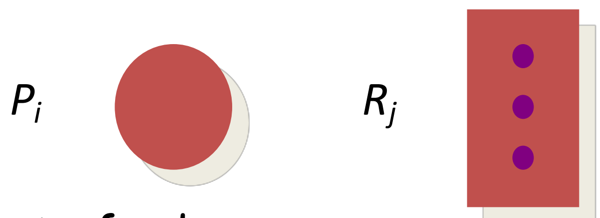
Deadlock: Definition



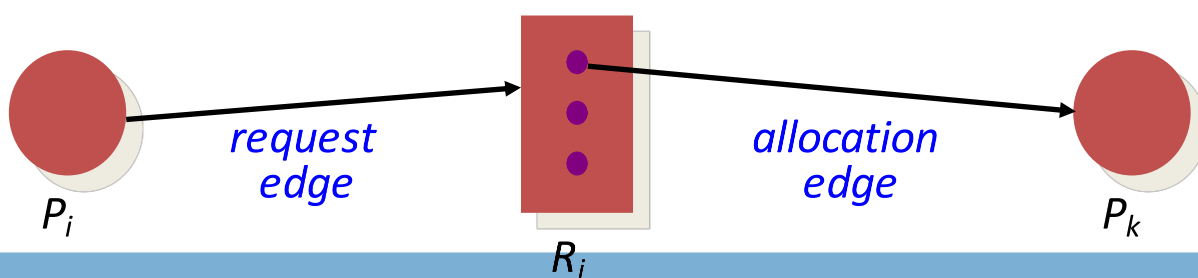
- A set of processes is deadlocked when every process in the set is waiting for an event that can only be generated by some process in the set
- Starvation vs. deadlock
  - Starvation: threads wait indefinitely (e.g., because some other thread is using a resource)
  - Deadlock: circular waiting for resources
  - Deadlock ➔ starvation, but not the other way

Resource Allocation Graph

- Basic components of any resource allocation problem
  - Processes and resources
- Model the state of a computer system as a directed graph
  - $G = (V, E)$
  - V = the set of vertices = $\{P_1, \ldots, P_n\} \cup \{R_1, \ldots, R_m\}$



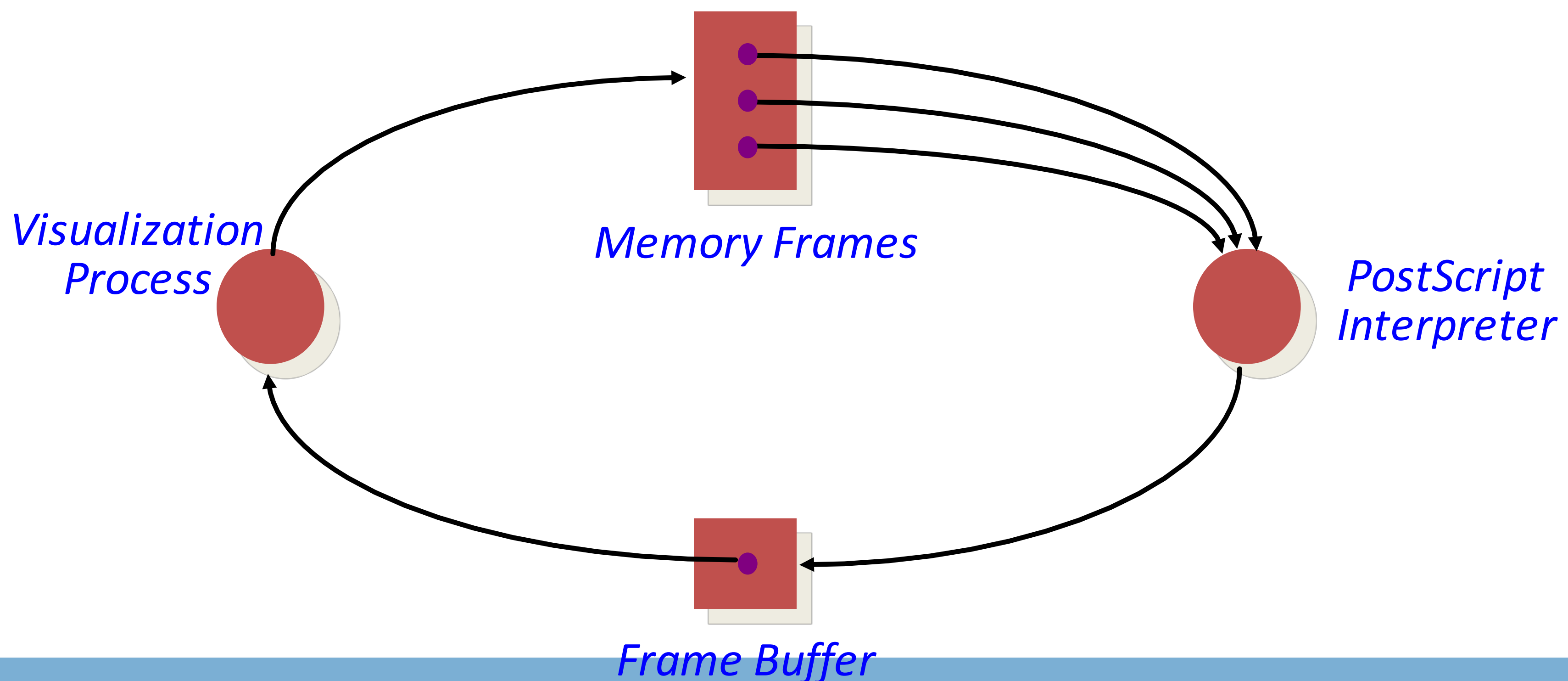
➢ E = the set of edges =
{*edges from a resource to a process*} $\cup$
{*edges from a process to a resource*}



Resource Allocation Graph: Example

- A PostScript interpreter that is waiting for the frame buffer lock and a visualization process that is waiting for memory

V = {*PS interpret*, *visualization*} ∪ {*memory frames*, *frame buffer lock*}

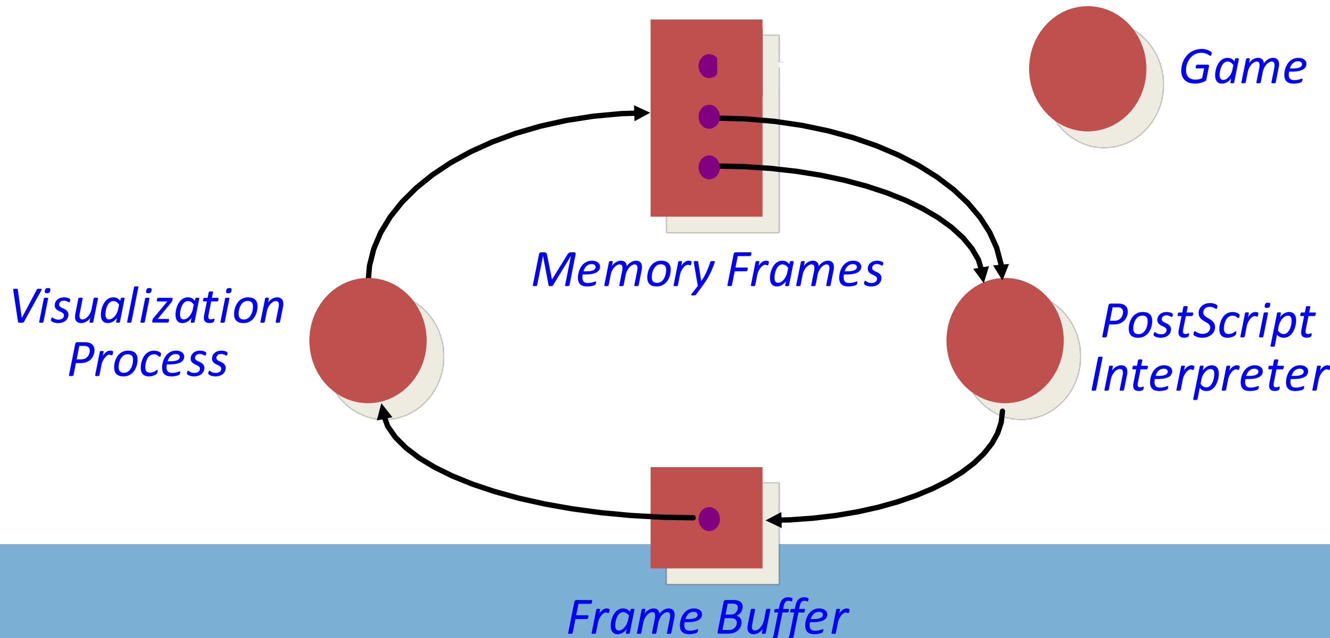


Resource Allocation Graph & Deadlock

- Theorem: *If a resource allocation graph does not contain a cycle then no processes are deadlocked*

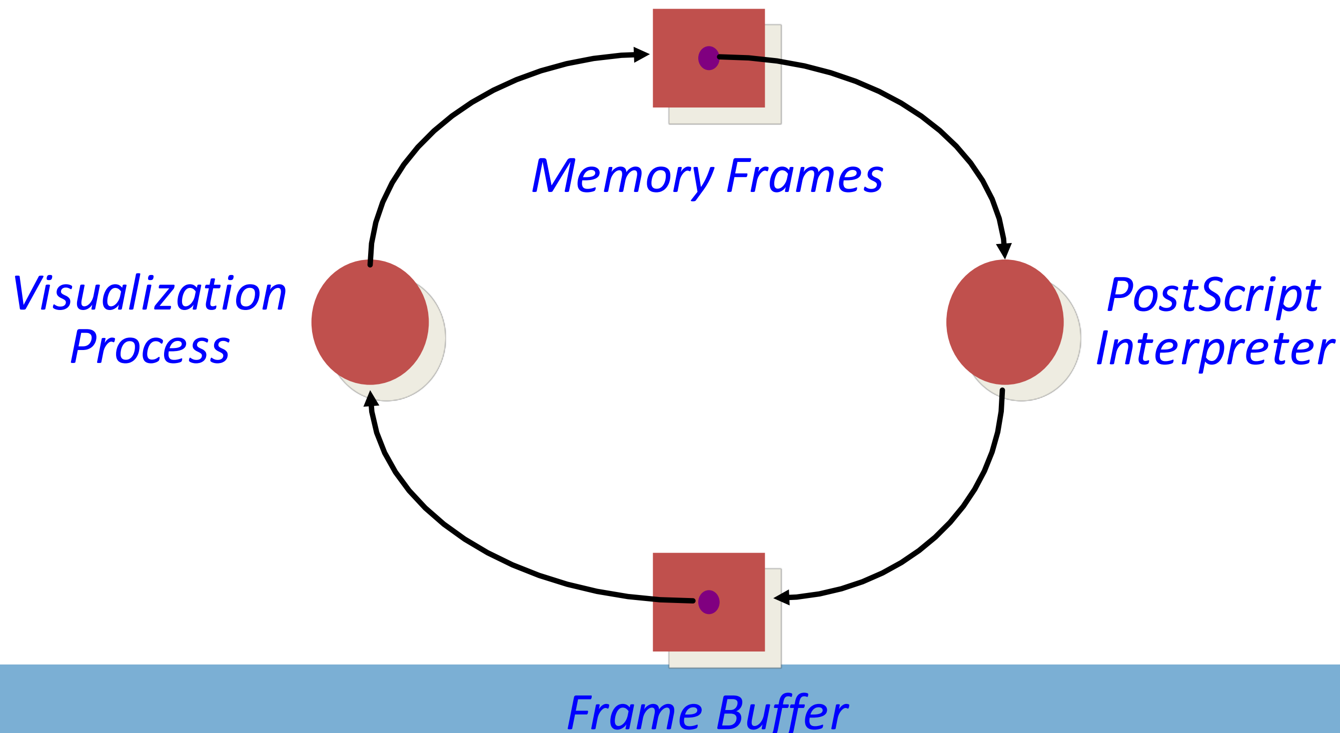
A cycle in a *RAG is* a necessary condition for deadlock

Is the existence of a cycle a sufficient condition?

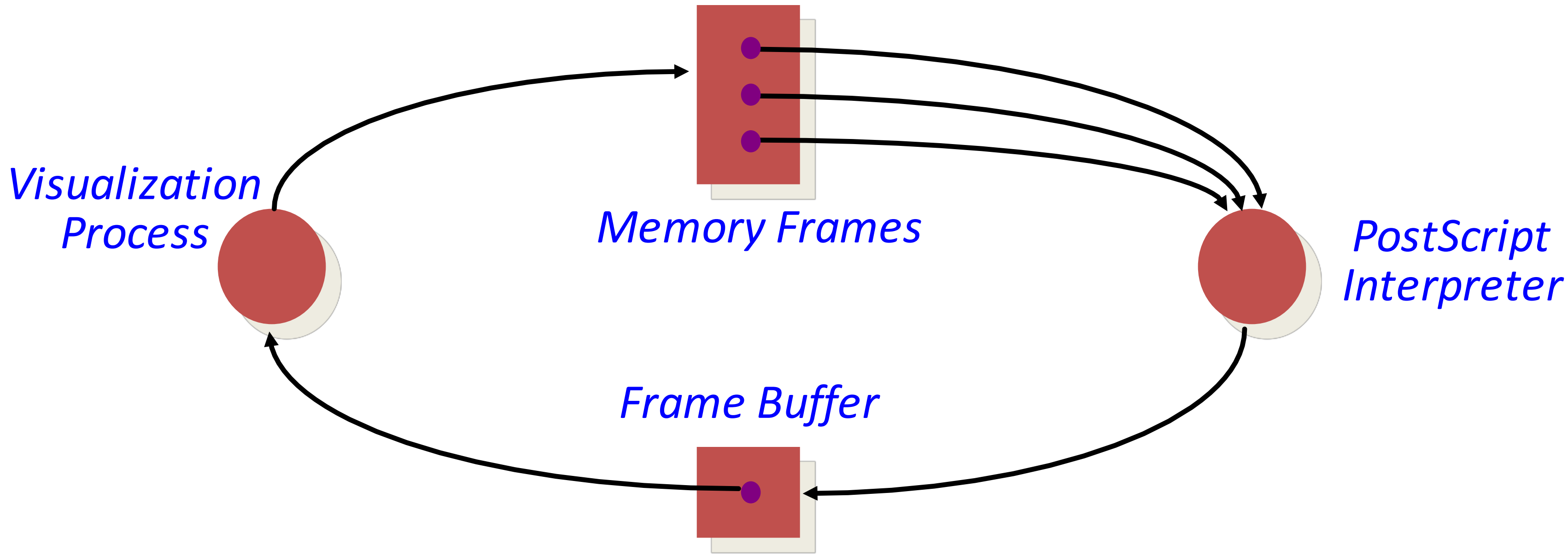


Resource Allocation Graph & Deadlock

- Theorem: *If there is only a single unit of all resources then a set of processes are deadlocked iff there is a cycle in the resource allocation graph*



An Operational Definition of Deadlock



- A set of processes are deadlocked *iff* the following conditions hold simultaneously
    1. Mutual exclusion is required for resource usage (serially useable)
    2. A process is in a "hold-and-wait" state
    3. Preemption of resource usage is not allowed
    4. Circular waiting exists (a cycle exists in the *RAG*)

Deadlock Prevention and/or Recovery

- Adopt some resource allocation protocol that ensures deadlock can never occur

  - Deadlock prevention/avoidance
    - Guarantee that deadlock will never occur
    - Generally breaks one of the following conditions:
      - Mutex
      - Hold-and-wait
      - No preemption
      - Circular wait *This is usually the weak link*

  - Deadlock detection and recovery
    - Admit the possibility of deadlock occurring and periodically check for it
    - On detecting deadlock, abort
      - Breaks the no-preemption condition
      - And non-trivial to restore all invariants

What does the RAG for a lock look like?

Deadlock Avoidance: Resource Ordering

- Recall this situation. How can we avoid it?

```
Producer1() {
  Lock(emptyBuffer)
  Lock(producerMutexLock)
   :
}
```

```
Producer2(){
  Lock(producerMutexLock)
  Lock(emptyBuffer)
   :
}
```

- Eliminate circular waiting by ordering all locks (or semaphores, or resoruces). All code grabs locks in a predefined order. Problems?
  - Maintaining global order is difficult, especially in a large project.
  - Global order can force a client to grab a lock earlier than it would like, tying up a resource for too long.
  - Deadlock is a global property, but lock manipulation is local.

Lock Ordering

- A program code convention
- Developers get together, have lunch, plan the order of locks
- In general, nothing at compile time or run-time prevents you from violating this convention
  - Research topics on making this better:
    - Finding locking bugs
    - Automatically locking things properly
    - Transactional memory

How to order?

- What if I lock each entry in a linked list. What is a sensible ordering?
  - Lock each item in list order
  - What if the list changes order?
  - Uh-oh! This is a hard problem
- Lock-ordering usually reflects static assumptions about the structure of the data
  - When you can't make these assumptions, ordering gets hard

Linux solution

- In general, locks for dynamic data structures are ordered by kernel virtual address
  - I.e., grab locks in increasing virtual address order
- A few places where traversal path is used instead

Lock ordering in practice
From Linux: fs/dcache.c

```
void d_prune_aliases(struct inode *inode) {
        struct dentry *dentry;
        struct hlist_node *p;
restart:
        spin_lock(&inode->i_lock);
        hlist_for_each_entry(dentry, p, &inode
                spin_lock(&dentry->d_lock);
                if (!dentry->d_count) {
                        __dget_dlock(dentry);
                        __d_drop(dentry);
                        spin_unlock(&dentry->d_lock);
                        spin_unlock(&inode->i_lock);
                        dput(dentry);
                        goto restart;
                }
                spin_unlock(&dentry->d_lock);
        }
        spin_unlock(&inode->i_lock);
}
```

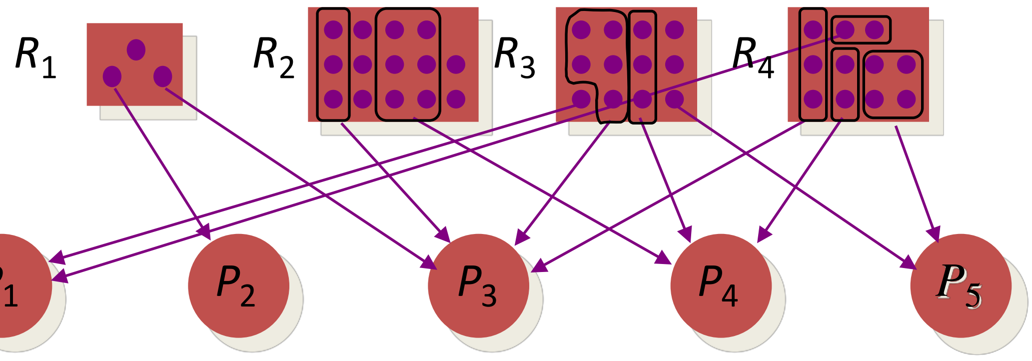
Care taken to lock inode before each alias

Inode lock protects list; Must restart loop after modification

mm/filemap.c lock ordering

```
/*
 * Lock ordering:
 *  ->i_mmap_lock               (vmtruncate)
 *    ->private_lock            (__free_pte->__set_page_dirty_buffers)
 *      ->swap_lock             (exclusive_swap_page, others)
 *        ->mapping->tree_lock
 *  ->i_mutex
 *    ->i_mmap_lock             (truncate->unmap_mapping_range)
 *  ->mmap_sem
 *    ->i_mmap_lock
 *      ->page_table_lock or pte_lock   (various, mainly in memory.c)
 *        ->mapping->tree_lock  (arch-dependent flush_dcache_mmap_lock)
 *  ->mmap_sem
 *    ->lock_page               (access_process_vm)
 *  ->mmap_sem
 *    ->i_mutex                 (msync)
 *  ->i_mutex
 *    ->i_alloc_sem             (various)
 *  ->inode_lock
 *    ->sb_lock                 (fs/fs-writeback.c)
 *    ->mapping->tree_lock      (__sync_single_inode)
 *  ->i_mmap_lock
 *    ->anon_vma.lock           (vma_adjust)
 *  ->anon_vma.lock
 *    ->page_table_lock or pte_lock     (anon_vma_prepare and various)
 *  ->page_table_lock or pte_lock
 *    ->swap_lock               (try_to_unmap_one)
 *    ->private_lock            (try_to_unmap_one)
 *    ->tree_lock               (try_to_unmap_one)
 *    ->zone.lru_lock           (follow_page->mark_page_accessed)
 *    ->zone.lru_lock           (check_pte_range->isolate_lru_page)
 *    ->private_lock            (page_remove_rmap->set_page_dirty)
 *    ->tree_lock               (page_remove_rmap->set_page_dirty)
 *    ->inode_lock              (page_remove_rmap->set_page_dirty)
 *    ->inode_lock              (zap_pte_range->set_page_dirty)
 *    ->private_lock            (zap_pte_range->__set_page_dirty_buffers)
 *  ->task->proc_lock
 *    ->dcache_lock             (proc_pid_lookup)
 */
```

Deadlock Recovery



- Abort all deadlocked processes & reclaim their resources
- Abort one process at a time until all cycles in the *RAG* are eliminated
- Where to start?
  - Select low priority process
  - Processes with most allocation of resources
- Caveat: ensure that system is in consistent state (e.g., transactions)
- Optimization:
  - Checkpoint processes periodically; rollback processes to checkpointed state

Common in Databases; Hard in General-Purpose Apps

Deadlock Avoidance: Banker's Algorithm

- Examine each resource request and determine whether or not granting the request can lead to deadlock

Define a set of vectors and matrices that characterize the current state of all resources and processes

- *resource allocation state matrix*

  $Alloc_{ij}$ = the number of units of resource j held by process i

- *maximum claim matrix*

  Max_{ij} = the maximum number of units of resource j that the process i will ever require simultaneously

- *available vector*

  $Avail_j$ = the number of units of resource j that are unallocated

$$
\begin{array}{c|ccccc|}
 & R_1 & R_2 & R_3 & \dots & R_r \\
P_1 & n_{1,1} & n_{1,2} & n_{1,3} & \dots & n_{1,r} \\
P_2 & n_{2,1} & n_{2,2} & & & \\
P_3 & n_{3,1} & & \ddots & & \vdots \\
\vdots & \vdots & & & & \\
P_p & n_{p,1} & & & & \\
 & & & \dots & & n_{p,r}
\end{array}
$$

$$\langle n_1, n_2, n_3, \dots, n_r \rangle$$

Dealing with Deadlock

- What are some problems with the banker's algorithm?
  - Very slow $O(n^2m)$
  - Too slow to run on every allocation. What else can we do?

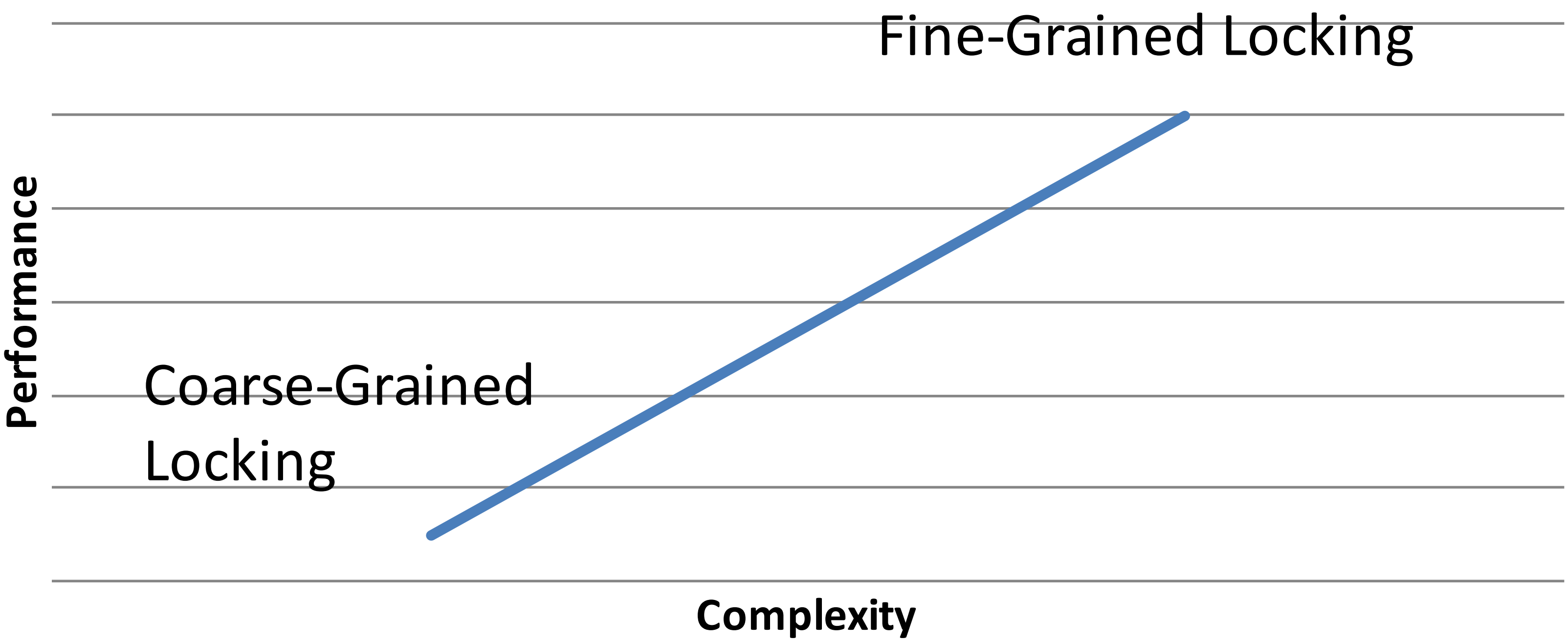
- Deadlock prevention and avoidance:
  - Develop and use resource allocation mechanisms and protocols that prohibit deadlock

- Deadlock detection and recovery:
  - Let the system deadlock and *then* deal with it
    - Detect that a set of processes are deadlocked
    - Recover from the deadlock

Summary and Editorial

- Deadlock is one difficult issue with concurrency
- Lock ordering is most common solution
  - But can be hard:
    - Different traversal paths in a data structure
    - Complicated relationship between structures
  - Requires thinking through the relationships in advance
- Other solutions possible
  - Detect deadlocks, abort some programs, put things back together (common in databases)
    - Transactional Memory
  - Banker's algorithm

Current Reality



- Unsavory trade-off between complexity and performance scalability