Overview

- Today: distributed shared memory, starting from some background on memory sharing
- Memory sharing for a single machine
  - Threads and processes
- Memory sharing for different machines
  - Threads and processes

Why Shared Memory?

- For sharing data
- There are two strategies for data sharing.
  - Message passing
  - Shared memory
- Message passing
  - Send/receive primitives
  - Explicit sharing \(\Rightarrow\) no synchronization (locks) necessary
- Shared memory
  - Memory read/write primitives (in your code, you could use regular variables)
  - Typically requires explicit synchronization (locks)
- Which is better?
  - Depends on your use case.
  - Multiple writers: perhaps message passing
  - (Mostly) read-only data: shared memory

Memory Sharing for Threads

- Threads belong to a single process, so all threads share the same memory address space.
- E.g., Java threads

```java
class MyThread extends Thread {
    HashMap hm;
    MyThread(HashMap _hm) {
        this.hm = _hm;
    }
    public void run() {
        hm.put(key, value);
    }
}
HashMap hashMap = new HashMap();
MyThread mt0 = new MyThread(hashMap); // hashMap is shared
MyThread mt1 = new MyThread(hashMap);
mt0.start();
mt1.start();
```

Memory: Threads vs. Processes

- For threads, there’s no special mechanism necessary to share memory.
  - Note: Languages like Java provide constructs to create thread-specific variables because by default memory is shared across different threads.
  - ThreadLocal for Java: if a shared object has a ThreadLocal variable, it will be specific to each thread.
- But, a process has its own address space, so by default, different processes do not share memory.
- Processes (on the same machine) can share memory regions with support from their OS.

Shared Memory on a Single Machine

- Shared memory is part of IPC (Inter-Process Communication).
  - What are other IPC mechanisms?
    - Files, (domain) sockets, pipes, etc.
- Shared memory API (POSIX C)
  - `shm_open()`: create and open a new object, or open an existing object. The call returns a file descriptor.
  - `mmap()`: map the shared memory object into the virtual address space of the calling process.
  - `...` and others
- Semaphore API (POSIX C)
  - `sem_open()`: initialize and open a named semaphore
  - `sem_wait()`: lock a semaphore
  - `sem_post()`: unlock a semaphore
  - `...` and others
Shared Memory Example* (in C)

```c
int main() {
    const char *name = "shared"; // shared with other processes
    int shm_fd;
    void *ptr;

    /* create the shared memory segment. name is shared. */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    ...

    /* now map the shared memory segment in the address space of
     * the process */
    ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

    sprintf(ptr, "%s", message0);
    return 0;
}
```

*Adapted from http://www.os-book.com*

---

Shared Memory Implementation

![Diagram](image)

- VPFN: Virtual page frame number
- PFN: Physical page frame number
- Adapted from http://tldp.org/LDP/mm/memory.html

---

Shared Memory Use Case: Android

- All apps need framework API libraries, Java VM, etc.
  - Too expensive if all app processes have them in their memory space individually.
- Zygote: A process that starts everything else.
  - All app processes share memory with Zygote.

![Diagram](image)

---

CSE 485/586 Administrivia

- PA3 grades will be posted today.
- PA4 deadline: 5/10
  - Please start early. The grader takes a long, long time.
- Survey & course evaluation
  - Survey: https://forms.gle/eg1wHN2G8S6GVz3e9
  - Course evaluation: https://www.smartevals.com/login.aspx?s=buffalo
- If both have 80% or more participation,
  - For each of you, I’ll take the better one between the midterm and the final, and give the 30% weight for the better one and the 20% weight for the other one.
  - (Currently, it’s 20% for the midterm and 30% for the final.)
- No recitation today; replaced with office hours

---

Distributed Shared Memory

- We will discuss two cases.
  - DSM for processes
  - DSM for threads
- DSM for processes: different processes running on different machines sharing a memory page.
- The shared memory page is replicated and synchronized across different machines.
  - However, replication is not the goal (e.g., we’re not keeping replicas to deal with failures).
- A generic way of doing this is at the OS layer.
  - Similar to the diagram on slide #8, but with processes on different machines

---

DSM Synchronization Options

- Write-update
  - A process updates a memory page.
  - The update is multicast to other replicas.
  - The multicast protocol determines consistency guarantees (e.g., FIFO-total for sequential consistency).
  - Reads are cheap (always local), but writes are costly (always multicast).
- Write-invalidate
  - Two states for a shared page: read-only or read & write
    - Read-only: the memory page is potentially replicated on two or more processes/machines.
    - Read & write: the memory page is exclusive for the process (no other replica)
  - If a process intends to write to a read-only page, an invalidate request is multicast to other processes.
  - Later writes can take place without communication (cheap).
  - Writes are only propagated when there’s a read by another process (cheap for write, costly for read).
  - But a write can be delayed by invalidation (costly for write).
Write Invalidate Protocol Example

- Note: R fault and W fault can occur at any process

Example System: Ivy

- Implements a write-invalidation protocol
  - Owner of a page: the process with the most up-to-date
  - Copyset of a page: the processes with a replica
  - A centralized manager maintains ownership info.

Granularity Problem

- Let’s assume that we operate at the page-level.
  - (But other implementations also have similar problems.)
  - Just as a reference, a Linux memory page is 4KB.
- Problem
  - When two processes (on two different machines) share a page, it doesn’t always mean that they share everything on the page.
  - E.g., one process reads from and writes to a variable X, while the other process reads from and writes to another variable Y. If they are in the same memory page, the processes are sharing the page.

Granularity Problem

- True sharing
  - Two processes share the exact same data.
- False sharing
  - Two processes do not share the exact same data, but they access different data from the same page.

- False sharing problems
  - Write-invalidate: unnecessary invalidations
  - Write-update: unnecessary data transfers

Granularity Problem

- Bigger page sizes
  - Better handling for updates of large amounts of data (good)
  - Less management overhead due to a smaller number of units/pages to handle (good)
  - More possibility for false sharing (bad)
- Smaller page sizes
  - The opposite of the above
  - If there is an update of a large amount of data, it’ll be broken down to many small updates, which leads to more network overhead (bad)
  - A smaller page size means more pages, which leads to more management overhead, i.e., more tracking of reads and writes (bad)
  - Less possibility of false sharing (good)

Thrashing

- Thrashing could happen with write-invalidate protocols.
- Thrashing is said to occur when DSM spends an inordinate amount of time invalidating and transferring shared data compared with the time spent by application processes doing useful work.
- This occurs when several processes compete for a data item or for falsely shared data items.
Thrashing

• Common scenario: producer-consumer pattern
  – Data is produced by a process and used by another process.
  – The producer will keep invalidating the consumer & the consumer will keep transferring data from the producer.
  – Write-update is better for this pattern.
• Solutions to thrashing
  – Timeslicing: once a process gains a write access to a page, it retains it for a period of time. Other processes’ read/write requests are buffered during that period.

DSM for Threads

• Memory sharing among threads on different machines.
• Use case: code (thread) offloading from a smartphone to a server
  – Low-power smartphones augmented by high-power servers (computation & energy)
  – It’s done already (cloud backend), but DSM allows it without any programmer effort.

Example: Comet*

• Comet allows thread offloading for Android apps in Java
• Comet synchronizes the entire Java VM state.

Java Code Execution Background

• Memory: program code, stack, heap, & CPU state
  – Stack & heap
    – Generally, the program stack handles statically allocated objects & method call return addresses.
    – The heap is used for dynamically allocated objects.
    – public class Ex {
      –   public void method() {
      –       int i = 0; // stack
      –       HashMap hm = new HashMap(); // heap
      –   }
    }
• CPU state
  – Android Java VM uses registers for instruction execution.
  – The program counter (PC) points to the next instruction to execute.
• For program execution, Java VM has an execution loop,
  – Fetches the next instruction that the PC points to.
  – Executes the new instruction
  – While executing, it uses registers, the stack, and the heap.

Comet Thread Migration

• Comet completely synchronizes VMs on both sides (phone & server).
  – In Java, everything you need for program execution is stored in memory.
  – Program code, stack, heap, & CPU state
  – DSM can synchronize these.
• Any side can execute a thread, since they both know everything necessary for program execution.
  – The PC is synchronized, so both sides know the next instruction to execute.
  – The registers are synchronized, so they both know the CPU state.
  – The stack & the heap are synchronized, so they know the memory state.

Summary

• Memory sharing among threads
  – By default, they share the same address space
• Memory sharing among processes
  – Shared memory API & semaphore API
  – Virtual-physical memory mapping implements this.
• Memory sharing across machines
  – Write-update
  – Write-invalidate
• Memory sharing across threads on different machines
  – Use case: code offloading

*https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gordon

https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gordon
Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).