Logistics

- Proj 3 Due on August 12
  - Milestone coming up on Tuesday
- HW3 Due on August 13
  - Last Tuesday of the class
- Special Topics Lecture on 8/13
  - Suggest ideas on Piazza thread (currently pinned)
Recall: Misses with Demand Paging

• Compulsory Misses: Prefetching
• Capacity Misses: Can't quickly add more RAM
• Conflict Misses: Don't technically occur, RAM acts as fully associative cache

• Comes down to having a good page replacement policy
Recall: Simple Page Replacement Policies

• Random

• FIFO (First In/First Out): Evict the oldest page (resident in memory the longest)
  • Bad: Age of a page does not necessarily reflect its utility

• Least Recently Used: Evict page whose last access was farthest in the past
  • Not practical to implement in this setting
Recall: Clock Algorithm

- *Used* bit per page
- Upon page fault: Scan through pages, reset the *used* bit back to 0
- Treat pages as circular sequence ("clock") – our current position is the "clock hand"
- Replacement candidates: Pages *not used since previous scan* (*Used* bit already 0)
- What if we scan around and all used bits are set?
  - Just loops around, reverts to FIFO
Recall: \(N^{th}\) Chance Clock Algorithm

- Replace the single bit (used since last cycle) with a counter – cycles since last reference
- Reset counter to 0 if page used, otherwise increment
- Candidate for eviction if counter > \(N\)
- Tradeoff: Larger \(N\) means better LRU approximation, but more scanning involved
Recall: Second-Chance List Algorithm

- Split memory in two: Active list (RW), SC list (Invalid)
- If we access a page on SC list, page fault occurs
  - Respond by moving to front of active list
- Replace a "victim" from front of SC list
Recall: Thrashing

- No individual process has enough pages to make steady progress, runs a little bit then page fault
- Result: Low CPU utilization
  - Every process waiting on IO to bring pages into memory
Reverse Page Mapping: "Coremap"

• Mapping from physical page frame # to all of its mappings in virtual address spaces

• Example uses:
  • Scanning accessed/dirty bits of all of frame's PTEs
  • Marking a page as not present in all PTEs that refer to it when evicting the page to disk
Coremap

Physical Memory

Frame 1

Frame 2

Frame 3

\vdots

Frame n

Page Table Entry

Proc. 1

Page Table Entry

Proc. 1

Page Table Entry

Proc. 1

Page Table Entry

Proc. 2
Loading Executable Into Memory

• View so far: OS copies each segment into memory
• Then set up registers, jump to start location
• One (very conservative) method: Every page in address space is backed by disk
• Just allocate space on disk, let a page fault trigger a load into memory
New View: Create Address Space

- Note that we do not need an extra copy of read-only data already contained in a file
- Executable code, memory mapped files (soon)
New View: Create Address Space

- User page table maps entire virtual address space
  - One per process, distinguishes present from absent pages
- OS needs to store mapping from virtual page to disk location
New View: Create Address Space

disk (huge, TB)

VAS – per proc.

kernel

stack

heap

data

code

Backering Store

Cache

memory

user page frames

user pagetable

kernel code & data
Provide Backing Store for VAS

disk (huge, TB)

stack
heap
data
code

stack
heap
data
code

stack
heap
data
code

VAS 1

VAS 2

PT 1

PT 2

memory

user page frames
user pagetable
kernel code & data
A Page Fault

disk (huge, TB)

stack

heap

data

code

VAS 1

kernel

stack

heap

data

code

VAS 2

kernel

stack

heap

data

code

PT 1

memory

user

page frames

user pagetable

kernel code & data

active process & PT
A Page Fault: Find and Start Load
A Page Fault: Switch During IO
On Page Fault: Update PTE
Eventually Reschedule Faulting Thread

disk (huge, TB)

memory

kernel code & data

user page frames

user pagetable

kernel code & data

active process & PT
Memory-Mapped IO

• IO so far: Explicit transfer between buffers in process address space to regions of a file
• Overhead: multiple copies in memory, syscalls

• Alternative: Map file directly into an empty region of a process address space
  • Implicitly page in file when we read it
  • Write to file and eventually page it out
• Executable file is treated this way when we execute a process!
Using Paging to `mmap` Files

Process instruction

virtual address

MMU

page#

PT

frame#

offset

File

scheduler

Operating System

Page Fault Handler

Read File contents from memory!

create PT entries for mapped region as “backed” by file

mmap() file to region of VAS

exception

Page Fault Handler

retry

physical address

Create PT entries for mapped region as “backed” by file

file to region of VAS

Create PT entries for mapped region as “backed” by file

file to region of VAS

file to region of VAS

file to region of VAS
mmap system call

- May map specific region or let the system find one for you
  - Tricky to know where a free region even would be…
**mmap Example**

```c
#include <sys/mman.h> /* also stdio.h, stdlib.h, string.h, fcntl.h, unistd.h */

int something = 162;

int main (int argc, char *argv[]) {
    int myfd;
    char *mfile;

    printf("Data at: %16lx\n", (long unsigned) &something);
    printf("Heap at : %16lx\n", (long unsigned) malloc(1));
    printf("Stack at: %16lx\n", (long unsigned) &mfile);

    /* Open the file */
    myfd = open(argv[1], O_RDWR | O_CREAT);
    if (myfd < 0) { perror("open failed!");exit(1); }

    /* map the file */
    mfile = mmap(0, 10000, PROT_READ|PROT_WRITE, MAP_FILE|MAP_SHARED, myfd, 0);
    if (mfile == MAP_FAILED) { perror("mmap failed"); exit(1); }
    printf("mmap at : %16lx\n", (long unsigned) mfile);
    puts(mfile);
    strcpy(mfile+20,"Let's write over it");
    close(myfd);
    return 0;
}
```

$ ./mmap test

```
Data at:              105d63058
Heap at :            7f8a33c04b70
Stack at:           7fff59e9db10
mmap at :           105d97000
This is line one
This is line two
This is line three
This is line four
```

$ cat test

```
This is line one
This is line two
This is line three
This is line four
```

Let's write over it
Sharing through Mapped Files

- Instructions
- Data
- Heap
- Stack
- OS

VAS 1
- 0x000…
- Memory
- File
- VAS 2
- 0x000…
- Stack
- OS
- 0xFFF…
32-bit x86 Linux Memory Layout

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

Stack (grows down)

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

Heap

BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;

Data segment
Static variables initialized by the programmer. Example: static char *gonzo = "God’s own prototype";

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

Program break
brk
start_brk
Random brk offset

End data
start_data
end_code

0x00000000
0x08048000

0xc0000000 == TASK_SIZE
Random stack offset
RLIMIT_STACK (e.g., 8MB)
Random mmap offset

1GB

3GB
32-bit x86 Linux Memory Layout

- Memory Mapped Files
- Extra stacks for multithreaded processes
- Shared Libraries
- Some Memory Allocations
32-bit x86 Linux Memory Layout

ASLR: Address Space Layout Randomization
Make it harder to exploit bugs to compromise system
See a security class for more...
32-bit x86 Linux Memory Layout

- **Memory Mapped Files**
- **Extra stacks for multithreaded processes**
- **Shared Libraries**
- **Some Memory Allocations**

**Kernel space**: User code CANNOT read from or write to these addresses, doing so results in a Segmentation Fault.

**Stack (grows down)**

**Memory Mapping Segment**: File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

**Heap**

**BSS segment**: Uninitialized static variables, filled with zeros. Example: static char *userName;

**Data segment**: Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;

**Text segment (ELF)**: Stores the binary image of the process (e.g., /bin/gonzo)
32-bit x86 Linux Memory Layout

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

Stack (grows down)

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

1GB

Heap

BSS segment
Uninitialized static variables, filled with zeros. Example: static char *userName;

3GB

Data segment
Static variables initialized by the programmer. Example: static char *gonzo = “God’s own prototype”;

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)

Protection bits: pages can't be accessed in user mode

Why? Faster than changing address space on every switch to kernel mode

Kernel mapped into every process's address space
Linux Virtual memory map

32-Bit Virtual Address Space

0x00000000

User Addresses

Kernel Addresses

0x00000000

0x00000000

0x00000000

0x00000000

1GB

896MB

Physical

0xC0000000

3GB Total

0x00000000

0xFFFF800000000000

"Canonical Hole"

64 TiB

Physical

0xFFFFF80000000000

128 TiB

0xFFFFF80000000000

"Canonical Hole"

64 TiB

Empty Space

User Addresses

User Addresses

Kernel Addresses

Kernel Addresses

0xFFFFFFFF

0xFFFFFFFF

0xFFFFFFFF

0xFFFFFFFF

128 TiB

64 TiB

128 TiB

128 TiB

32-Bit Virtual Address Space

64-Bit Virtual Address Space
Linux Virtual memory map

32-Bit Virtual Address Space

- 0x00000000
- 0x00000000
- 0x00007FFFFFFFFFFF
- 0xFFFFFFFF

- 4GB Total
- 896MB
- Physical
- 1GB
- 0xC0000000

64-Bit Virtual Address Space

- 0x0000000000000000
- 0xFFFFFFFF80000000
- 128TiB
- "Canonical Hole"

One-to-One maps of "bottom" of Physical Memory

Kernel Addresses

Empty Space

User Addresses

User Addresses

Kernel Addresses

User Addresses

Kernel Addresses
Linux Virtual memory map

32-Bit Virtual Address Space

64-Bit Virtual Address Space

Kernel Addresses

User Addresses

Can temporarily map higher physical addresses

Empty Space

"Canonical Hole"

128TiB

896MB

3GB Total

1GB

Physical

0x00000000

0xC0000000

0xFFFFFFFF

0x0000000000000000

0x00007FFFFFFF

0xFFFF800000000000

0xFFFFFFFFFFFFFFFF

0x0000000000000000

0x00007FFFFFFF

0xFFFF800000000000

0xFFFFFFFFFFFFFFFF

3GB Total

128TiB

896MB

32

64

Bit Virtual Address Space

64

Bit Virtual Address Space
Possible to inspect contents of kernel memory if it's mapped into address space (even as user!)

Fix: Kernel Page Table Isolation
- Use entirely different page tables when in user mode vs. when in kernel mode

Problem: Address space change whenever an interrupt or syscall occurs!
- Change page tables
- Flush TLB unless it is tagged

Reduced Performance, depends on syscall workload
Break
Interprocess Communication

• Thus far, we've said the following:
• Hard to cooperate across processes because they're inherently isolated (separate addr. spaces)
  • But this is good for protection

• Easy to cooperate among threads because they share an address space
  • But this is bad for protection
  • Have to use synchronization primitives like locks
Interprocess Communication

• Allow two (or more) processes to exchange information with each other

• Why use this approach rather than multithreading?
  • Keep most of the benefits of process isolation
  • Expose processes to each other only through a *carefully structured* interface
    • Ex: [Google Chrome](https://www.google.com) Design
We've Already Seen Some IPC

1. Two processes share a file (e.g. both mmap it)
   • Needs some synchronization, must structure file layout
   • Con: Still involves entire kernel IO infrastructure
   • What if file is only temporary, needed while processes are running?

2. Open a socket to 127.0.0.1 (localhost)
   • Nice if we ever want to deploy process on remote machine later on
   • But lots of extra work: Packet/header assembly, checksum calculation, TCP ACKs, etc.
IPC Example: Pipes

• Use classic file-related syscalls: **read, write**

• But avoid the overhead of actually interacting with kernel IO subsystem

• Instead: writes/reads manipulate a buffer of memory maintained by the kernel
Pipe Syscall

PIPE(2)
Linux Programmer's Manual

NAME
   pipe, pipe2 - create pipe

SYNOPSIS
   
   #include <unistd.h>

   int pipe(int pipefd[2]);

DESCRIPTION
   pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe.

   Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.
Using Pipes – Common Pattern

• Process calls `pipe` with 2-int array
  • `array[0]`: Read-only
  • `array[1]`: Write-only

• `fork()`

• Each pipe fd is unidirectional – parent and child each close the "end" they don't need
Using Pipes

```c
int pipe_fds[2];
pid_t cpid;

pipe(pipe_fds);
cpid = fork();
if (cpid > 0) {
    close(pipe_fds[0]); // Close read-only end
    char *msg = "Hello, World!";
    write(pipe_fds[1], msg, 13);
    close(pipe_fds[1]);
    waitpid(cpid, NULL, 0);
} else {
    close(pipe_fds[1]); // Close write-only end
    char buf[1024];
    int nread = read(pipe_fds[0], &buf, 1023);
    buf[nread] = '\0';
    printf("Parent Sent: %s\n", buf);
    close(pipe_fds[0]);
}
```
Using Pipes

- Remember producer-consumer problem?
- Pipe's buffer has maximum size
  - Write to full pipe blocks until space available
  - Read from empty pipe blocks until data available
- Read from pipe with no writers returns 0
- Write to pipe with no readers prompts SIGPIPE
UNIX Domain Sockets

• Open a socket connection with a local process

• Use familiar write/read calls to communicate

• But don't incur usual overhead of networking

• Optimized for processes on same machine
Using Unix Domain Sockets

• Still need same sequence of syscalls: `socket, bind, listen, accept` to act as a server

• But socket address now corresponds to an object in local machine's filesystem
  • Specify path on bind

• Why this approach?
  • Filesystem gives us a namespace: any process can specify path on `connect` (doesn't need to be a child of server)
  • Filesystem enforces permissions
#include <sys/un.h>

...  
int fd;
struct sockaddr_un un;
un.sun_family = AF_UNIX;
strcpy(un.sun_path, "/home/oski/demo.socket");
fd = socket(AF_UNIX, SOCK_STREAM, 0);
bind(fd, (struct sockaddr*)&un, sizeof(un));
Many Other Forms of IPC

- Named Pipes (FIFOs): Pipe interface, but given a name in the file system (man 3 mkfifo)
- Named semaphores (man sem_open)
- Message Queues (man 7 mq_overview)
- And more…
Summary

• Alternate view of loading a program: allocate space on disk, load in pages only when needed

• Memory-Mapped IO: Map contents of file into virtual address space, store/load instead of read/write

• Inter-Process Communication: Structured sharing
  • Pipes: Read/write ordered, in-memory buffer
  • Unix Domain Sockets: Avoid networking overhead
C Concurrency and Synch.

• Standard approach: use pthreads, protect access to shared data structures
• Shared Memory Paradigm
• One pitfall: consistently unlocking a mutex

```c
int Rtn() {
    lock.acquire();
    ...
    if (exception) {
        lock.release();
        return errCode;
    }
    ...
    lock.release();
    return OK;
}
```
Other Languages and Threading

• Many other mainstream languages also focus on threads and shared memory

• But offer useful libraries and built-in features to make our lives easier
  • Thread management libraries
  • Thread pools
  • Safer lock management
  • Objects as monitors
# C++ Lock Guards

```cpp
#include <mutex>
int global_i = 0;
std::mutex global_mutex;

void safe_increment() {
  std::lock_guard<std::mutex> lock(global_mutex);
  ...
  global_i++;
  // Mutex released when 'lock' goes out of scope
}
```
Python with Keyword

• More versatile than we'll show here (can be used to close files, server connections, etc.)

```python
with lock:  # Automatically calls acquire()
    some_var += 1
...
# release() called however we leave block
```
Java Support for Synchronization

class Account {
    private int balance;

    public Account (int initialBalance) {
        balance = initialBalance;
    }

    public synchronized int getBalance() {
        return balance;
    }

    public synchronized void deposit(int amount) {
        balance += amount;
    }
}

• Every Java object has an associated lock for synchronization:
  • Lock is acquired on entry and released on exit from synchronized method
  • Lock is properly released if exception occurs inside synchronized method
Java Support for Synchronization

• Along with a lock, every object has a **single** condition variable associated with it

• To wait inside a synchronized method:
  • `void wait();`
  • `void wait(long timeout);`

• To signal while in a synchronized method:
  • `void notify();`
  • `void notifyAll();`
Up Next: Go

• "Goroutines": Lightweight, user-level threads

• Channels: Named message queues for communication among threads

• Key Idea: Prefer message passing over shared memory
Why this approach?

- Efficiency of a shared address space
- Tolerates many threads in one program
- Passing data through channels: no need for explicit synchronization
  - Sender *passes ownership* to receiver