Logistics

- Project 1 Milestone Due Wednesday
- Homework 1 Due Friday
- Project/HW “Party” Friday 3-5PM
  - Wozniak Lounge, Soda Hall
  - Full course staff will be there to help you
- Midterm Review
Recall: Starvation vs. Deadlock

- **Starvation:** A thread waits indefinitely
  - Ex: Low-priority threads, when other high-priority threads are always present

- **Deadlock:** Circular waiting for resources
  - A case of starvation
Recall: Four Requirements for Deadlock to Occur

1. **Mutual Exclusion**: One thread at a time can use a resource (not shareable)

2. **Hold and Wait**: Thread holding a resource waits to acquire another resource

3. **No Preemption**: Resources are released voluntarily, threads can’t steal instead of waiting

4. **Circular Wait**: There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads such that:
   - \( T_1 \) is waiting for a resource held by \( T_2 \)
   - \( T_2 \) is waiting for a resource held by \( T_3 \)
   - \( \ldots \)
   - \( T_n \) is waiting for a resource held by \( T_1 \)
Recall: Banker’s Algorithm

• Need to know each thread’s maximum resource requirements in advance

• Block a thread from getting a resource unless system would remain in a safe state
  • There is an ordering of threads $T_1, T_2, \ldots, T_n$ such that running $T_1$ to completion, $T_2$ to completion, and so on would not deadlock

• Use modified version of deadlock detection algo.
  • Pretend each resource request has been satisfied
  • Pretend each running thread is requesting its maximum
  • If no deadlock, then an ordering exists
Banker’s Algorithm

// Assign each resource an index i
Avail[i] = Free[i] for all i
Add all nodes to UNFINISHED

do {
    done = true;
    for each node in UNFINISHED {
        if (Max_{node}[i] - Alloc_{Node}[i] <= Avail[i] for all i) {
            remove node from UNFINISHED
            Avail[i] += Allocated_{node}[i] for all i
            done = false
        }
    }
}
} until(done)

Nodes left in UNFINISHED => Unsafe State
Recall: I/O & Storage Layers

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

- streams
- handles
- registers
- descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA
Recall: Streams vs. File Descriptors

- Streams are **buffered in user memory**:  
  ```c
  printf("Beginning of line ");
  sleep(10); // sleep for 10 seconds
  printf("and end of line\n");
  ```  
  Prints out **everything at once**

- Operations on file descriptors are **visible immediately**  
  ```c
  write(STDOUT_FILENO, "Beginning of line ", 18);
  sleep(10);
  write(STDOUT_FILENO, "and end of line \n", 16);
  ```  
  Outputs "Beginning of line" 10 seconds earlier
Stream API Review

#include <stdio.h>

int main(void) {
    FILE* f = fopen("output.txt", "w");
    fwrite("CS 61A\n", sizeof(char), 7, f);
    fseek(f, 0, SEEK_SET);
    fprintf(f, "CS 61B\n");
    fclose(f);

    f = fopen("output.txt", "a");
    fwrite("CS 61C\n", sizeof(char), 7, f);
    fclose(f);

    f = fopen("output.txt", "w");
    fprintf(f, "CS 162\n");
    fclose(f);
    return 0;
}

What does output.txt contain during the execution of this code?
I/O & Storage Layers

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams

handles

registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA
Recall: SYSCALL

- Usually: one syscall directly corresponds to one lower-level I/O operation
- System library puts parameters in registers, invokes special instruction to trap into kernel
Internal OS Open File

Description

- Internal data structure (*struct file* in Linux)
- Describes everything about the open file
- Looked up within PCB by file descriptor number

```c
746 struct file {
747     union {
748         struct llist_node fu_llist;
749         struct rcu_head fu_rcuhead;
750     } f_u;
751     struct path *f_path;
752     struct dentry *f_dentry;
753     struct inode *f_inode; /* caci */
754     const struct file_operations *f_op;
755     spinlock_t f_lock;
756     atomic_long_t f_count;
757     unsigned int f_flags;
758     fmode_t f_mode;
759     struct mutex f_pos_lock;
760     loff_t f_pos;
761     struct fown_struct f_owner;
762     const struct cred *f_cred;
763     struct file_ra_state f_ra;
764     u64 f_version;
765 #ifdef CONFIG_SECURITY
766     void (*f_security)();
767 #endif
768 #ifdef CONFIG_EPOLL
769     struct list_head f_ep_links;
770     struct list_head f_tfile_llink;
771 #endif /* ifdef CONFIG_EPOLL */
772 } __attribute__((aligned(4))); /* lest something weir
```
Recall: I/O & Storage Layers

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

Commands and Data Transfers

streams

handles

registers

descriptors

Disks, Flash, Controllers, DMA
```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```

- Read up to “count” bytes from “file” starting from “pos” into “buf”.
- Return error or number of bytes read.
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
In `fs/read_write.c`

```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```

Check if file has read methods
```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
    }
    inc_syscr(current);
    return ret;
}
```

- Check whether we can write to buf (e.g., buf is in the user space range)
- unlikely(): hint to branch prediction this condition is unlikely
File System: from syscall to driver

In fs/read_write.c

```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```

Check whether we read from a valid range in the file.
File System: from syscall to driver

In fs/read_write.c

ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!(file->f_op || file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
 ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
File System: from syscall to driver
ln fs/read_write.c

ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
In file system code:

```c
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos) {
    ssize_t ret;
    if (!(file->f_mode & FMODE_READ)) return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
        else
            ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
            fsnotify_access(file->f_path.dentry);
            add_rchar(current, ret);
        }
        inc_syscr(current);
    }
    return ret;
}
```

Update the number of read syscalls by “current” task (for scheduling purposes)
Lower Level Driver

- Associated with particular hardware device
- Registers / Unregisters itself with the kernel
- Handler functions for each of the file operations

```c
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, f1_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    [...]}
};
```
Device Driver

• Software module that interacts directly with hardware (issues commands)

• Provides a standard interface to the OS, same kernel I/O system can interact with different devices

• Two parts:
  1. "Top Half" accessed from system calls
     • open, read, write, ioctl, ...
     • Starts I/O
  2. "Bottom Half" invoked from interrupt handler
     • Gets input or transfers output when device is ready
     • Responsible for waking blocked threads when I/O is complete
Life Cycle of An I/O Request

**User Program**
- User makes a system call
  - **Top Half**
    - Device Driver
    - **Bottom Half**
      - Device Hardware

**Kernel I/O Subsystem**
- System call
  - if I/O request can already be satisfied
    - Transfer data
  - if no
    - Send request to device driver
      - Device Driver
        - Top Half
          - process request, issue commands to controller
            - Device Driver
              - Bottom Half
                - Device Hardware
                  - Monitor device, interrupt when I/O completed

**Return from system call**
- I/O completed, input data available, or output completed
  - User process
  - Return completion or error code
Communication Across the World Looks Like File IO

write(wfd, wbuf, wlen);

n = read(rfd, rbuf, rmax);

- Sockets: Connected queues over the Internet
  - How to open()? Filenames?
  - How are they connected in time?
**Request Response Protocol**

Example: Web browser and website

**Client (issues requests)**

write(rqfd, rqbuf, buflen);

**Server (performs operations)**

n = read(rfd, rbuf, rmax);

wait

n = read(resfd, resbuf, resmax);

write(wfd, respbuf, len);

service request

requests

responses
Client-Server Models

- Many clients accessing a common server
- All intelligence in the server
**Sockets**

- **Socket**: An abstraction of a network I/O queue
  - Mechanism for *inter-process communication*
  - Embodies one side of a communication channel
  - Same interface regardless of local or remote

- **First introduced in 4.2 BSD Unix**
  - Most operating systems (Linux, Mac OS X, Windows) provide this, even if they don’t copy rest of UNIX I/O
  - Standardized by POSIX
Sockets

• Looks just like a file with a **file descriptor**
  • **read** adds to queue, **write** removes from it
  • Bidirectional: one queue in each direction
  • Some operations do not work, e.g. **lseek**

• Same abstraction for any kind of network
  • Local (within same machine)
  • The Internet (TCP/IP, UDP/IP)
  • Things “no one” uses anymore (OSI, Appletalk, IPX, …)
Silly Echo Server

Client (issues requests)

Server (performs operations)

User

gets(fd, sndbuf, ...);

write(fd, buf, len);

wait

requests

responses

n = read(fd, buf,);

print

n = read(fd, rcvbuf,);

print

write(fd, buf,);
void client(int sockfd) {
    int n;
    char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
    getreq(sndbuf, MAXIN); /* prompt */
    while (strlen(sndbuf) > 0) {
        write(sockfd, sndbuf, strlen(sndbuf)); /* send */
        memset(rcvbuf,0,MAXOUT); /* clear */
        n=read(sockfd, rcvbuf, MAXOUT-1); /* receive */
        write(STDOUT_FILENO, rcvbuf, n); /* echo */
        getreq(sndbuf, MAXIN); /* prompt */
    }
}

void server(int consockfd) {
    char reqbuf[MAXREQ];
    int n;
    while (1) {
        memset(reqbuf,0,MAXREQ);
        n = read(consockfd,reqbuf,MAXREQ-1); /* Recv */
        if (n <= 0) return;
        n = write(STDOUT_FILENO, reqbuf, strlen(reqbuf));
        n = write(consockfd, reqbuf, strlen(reqbuf)); /* echo*/
    }
}
char *getreq(char *inbuf, int len) {
    /* Get request char stream */
    printf("REQ: ");          /* prompt */
    memset(inbuf,0,len);       /* clear for good measure */
    return fgets(inbuf,len,stdin); /* read up to a EOL */
}
Socket Creation and Connection

- File systems: permanent objects
  - Files exist independently of processes
- Sockets are transient, tied to particular processes (the two endpoints!)
- Creation and communication is more complex
- Form two-way byte streams between processes
  - Possibly worlds away
IP Namespaces

• Hostnames,
  • Ex: www.berkeley.edu
  • DNS Name, String

• IP Address
  • 169.229.216.200 (IPv4, 32-bit Integer)
  • 2607:f140:0:81::f (IPv6, 128-bit Integer)

• Port Number (Demultiplex different processes on same host)
  • 0-1023 typically reserved for administrator (superuser/sudo)
  • 0-49151 are “well known” numbers for specific services
    • 443 for HTTPS, 22 for SSH
  • 49152-65535 ($2^{15}+2^{14}$ to $2^{16}-1$) are “dynamic”/”private”
    • Automatically allocated by the OS
Socket Setup over TCP/IP

- Special kind of socket: **server socket**
  - Has file descriptor
  - Can’t read or write

- Two operations:
  1. `listen()`: Start allowing clients to connect
  2. `accept()`: Create a new socket for a particular client
Socket Setup over TCP/IP

• 5-Tuple identifies each connection:
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol (always TCP here)

  • Server port is “well known”
  • Where does client get its port number from?
  • Recall: dynamic/private port range
Sockets in concept

**Client**
- Create Client Socket
- Connect it to server (host:port)
  - Connection Socket
  - write request
  - read response
- Close Client Socket

**Server**
- Create Server Socket
- Bind it to an Address (host:port)
- Listen for Connection
- Accept syscall()
  - Connection Socket
  - read request
  - write response
- Close Connection Socket
- Close Server Socket
Client Protocol

char *hostname; char* portname;
int sockfd;
struct addrinfo *server;
struct hostent *server;
server = buildServerAddr(hostname, portname);

// Create a TCP socket
// server->ai_family: AF_INET (IPv4) or AF_INET6 (IPv6)
// server->ai_socktype: SOCK_STREAM (byte-oriented)
// server->ai_protocol: IPPROTO_TCP
sockfd = socket(server->ai_family, server->ai_socktype, server->ai_protocol)

// Connect to server on port
connect(sockfd, server->ai_addr, server->ai_addrlen);
// Carry out Client-Server protocol
client(sockfd);

/* Clean up on termination */
close(sockfd);
freeaddrinfo(server);
/* Create Socket to receive requests */
lstnsockfd = socket(server->ai_family, server->ai_socktype,
                    server->ai_protocol);

/* Bind socket to port */
bind(lstnsockfd, server->ai_addr, server->ai_addrlen);
/* Listen for incoming connections */
listen(lstnsockfd, MAXQUEUE);
while (1) {
    /* Accept incoming connection, obtaining a new socket for it */
    consockfd = accept(lstnsockfd, NULL, NULL);

    server(consockfd);

    close(consockfd);
}
close(lstnsockfd);
Aside: Fork and File Descriptors

```c
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC);
    pid_t cpid = fork();
    if (cpid == 0) {
        // Child process
        write(fd, "CS 61A\n", sizeof(char)*7);
    } else {
        waitpid(cpid, NULL, 0);
        write(fd, "CS 61B\n", sizeof(char)*7);
    }
    return 0;
}
```
Handling Multiple Connections

• One Option: Fork a process for each connection
  • Strong isolation between each connection
  • Can accept new connections while others are active
  • Very Expensive

• Second option: Spawn a thread for each connection
  • Or better yet, use a thread pool

• Third option: Event style
Summary: Sockets

• Abstraction of network I/O interface
  • Bidirectional communication channel
  • Uses file interface once established (read, write, close)

• Server setup:
  • socket, bind, listen, accept
  • read, write, close from socket returned by accept

• Client setup:
  • socket, connect
  • then read, write close

• Use gettaddrinfo to resolve names, addresses for bind and connect