An OS Design for Memory Constrained Platforms

Yacqub Mohamed, Robbert van Renesse

Agenda

- Overview of OS capable of running on constrained hardware
- Recap of traditional OS techniques to improve performance
- Why the techniques are undesirable on constrained hardware
- Motivation for multiplexing in constrained environment
- OS Design for Efficient Multiplexing in Constrained Environments

- Overview of OS capable of running on constrained hardware
- Recap of traditional OS techniques to improve performance
- Why the techniques are undesirable on constrained hardware
- Motivation for multiplexing in constrained environment
- OS Design for Efficient Multiplexing in Constrained Environments

EGOS 2000 is an educational OS built to work with constrained hardware

Earth Layer: Handles Hardware Specifics

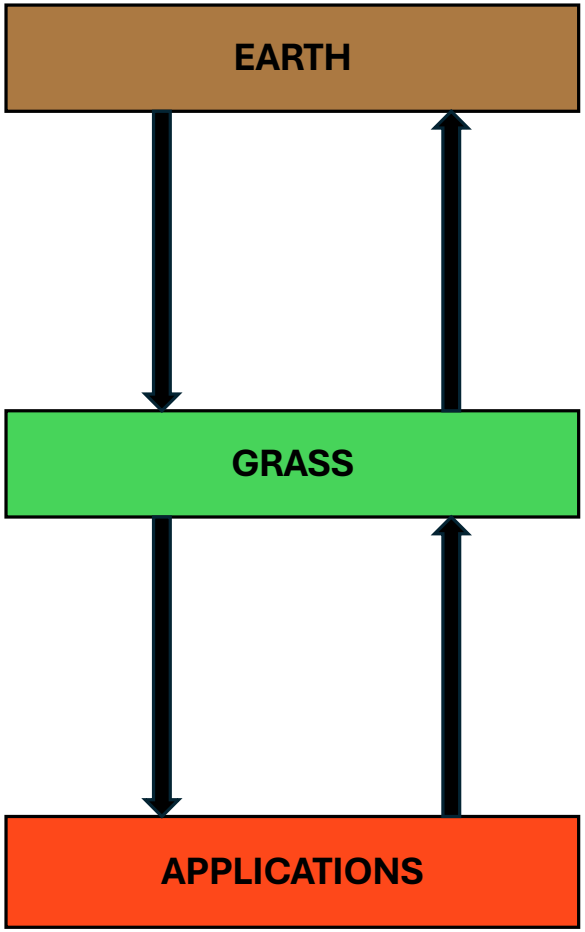
- Memory Management (Disk, Paging, Protections)
- TTY Interface & UART Device
- Timer

Grass Layer: Handles Hardware Abstractions

- Process Management & Scheduling
- System Calls
- Exception Handling

App Layer: Handles Interactions with Hardware Abstractions

- Creating and Killing Processes
- Reading/Writing Files & Directories
- Reading/Writing Keyboard Input



Classic Operating Systems Problem

Kernel Code in EGOS 2000 is Busy Waiting for I/O Devices

Keyboard Device Driver:

```
int tty_write(char* buf, uint len) {
    for (uint i = 0; i < len; i++) uart_putc(buf[i]);
}

int tty_read(char* buf, uint len) {
    is_reading = 1;
    for (uint i = 0; i < len - 1; i++) {
        for (c = -1; c == -1; uart_getc(&c));
        buf[i] = (char)c;
```

```
int uart_getc(int* c) {
    uint ch = REGW(UART0_BASE, UART0_RXDATA);
    return *c = (ch & (1 << 31))? -1 : (ch & 0xFF);
}

void uart_putc(int c) {
    while ((REGW(UART0_BASE, UART0_TXDATA) & (1 << 31)));
    REGW(UART0_BASE, UART0_TXDATA) = c;
}
```

SD Card Device Driver:

```
static void single_read(uint offset, char* dst) {
    /* Wait until SD card is not busy */
    while (recv_data_byte() != 0xFF);

    /* Send read request with cmd17 */
    char *arg = (void*)&offset;
    char reply, cmd17[] = {0x51, arg[3], arg[2], arg[1], arg[0], 0xFF};

    if (reply = sd_exec_cmd(cmd17))
        FATAL("SD card replies cmd17 with status 0x%.2x", reply);

    /* Wait for the data packet and ignore the 2-byte checksum */
    while (recv_data_byte() != 0xFE);
```

```
char send_data_byte(char byte) {
    while (REGW(SPI1_BASE, SPI1_TXDATA) & (1 << 31));
    REGB(SPI1_BASE, SPI1_TXDATA) = byte;

    uint rxdata;
    while ((rxdata = REGW(SPI1_BASE, SPI1_RXDATA)) & (1 << 31));
    return (char)(rxdata & 0xFF);
}

char recv_data_byte() { return send_data_byte(0xFF); }
```

```
[CRITICAL] Welcome to the egos-2000 shell!
→ /home/yunhao
```

Classic OS Problem

Why Busy Waiting is a Problem:

1. **Scheduling**
   - The OS wastes CPU time scheduling processes, such as the shell, that are waiting for input that has not been received yet
   - CPU time could instead go to other processes not waiting for input
2. **Power Consumption**
   - Even with no activity, the hardware is continually running and executing instructions that don't do anything meaningful

- Overview of OS capable of running on constrained hardware
- Recap of traditional OS techniques to improve performance
- Why the techniques are undesirable on constrained hardware
- Motivation for multiplexing in constrained environment
- OS Design for Efficient Multiplexing in Constrained Environments

Solution to Classic OS Problem

External Interrupts: Devices signal when an I/O operation completes

Why External Interrupts solve Busy Waiting (and general uses):

1. **Scheduling**
   - Processes that are waiting for input are not scheduled again until the device signals completion
   - Frees CPU time for other runnable processes
2. **Power Consumption**
   - While waiting for I/O activity, the operating system will **wait for interrupt (wfi)**
     - Instruction execution halts
     - Device signal resumes execution, able to interact with I/O devices
3. **Buffering**
   - Before processes interact with I/O Devices, kernel can buffer input on device signal
   - By the time a process attempts to read an I/O Device, data will already be available

- **How does the OS know which device the signal came from?**

External Interrupts: Platform Level Interrupt Controller (PLIC)

- **Maintains information on:**
  - Enabled Device Interrupts
  - Pending Device Interrupts
  - ID of Pending Interrupt Signal

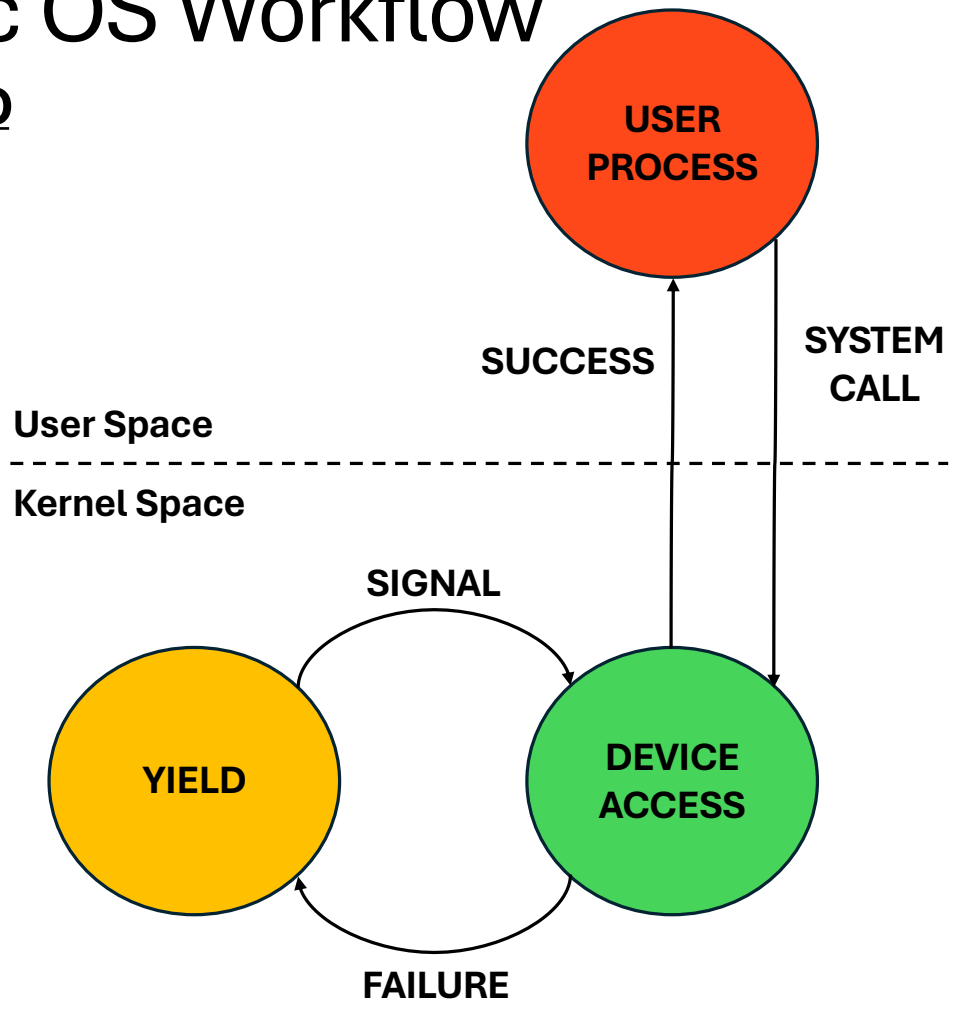
- **On External Interrupt:** Interrupt Handler Workflow
  - Read Claim Register for Pending Interrupt ID
  - Handle Device-Specific Interrupt
  - Write Claim Register with Pending ID to Disable Pending Interrupt

```c
int trap_external()
{
    int intr_cause = REGW(PLIC_CLAIM_BASE, 0);
    int rc;

    switch (intr_cause)
    {
    case PLIC_UART0_ID:
        rc = tty_handle_intr();
        break;
    default:
        rc = 0;
    }

    REGW(PLIC_CLAIM_BASE, 0) = intr_cause; // Interrupt Pending Bit is Cleared
    return rc;
}
```

External Interrupts: Classic OS Workflow

- **User Process makes System Call for I/O Device:**
  1. Process (in kernel context) attempts to access I/O Device
     - **Success:** Return to user space
     - **Failure:** Yield
  2. On Failure, the OS blocks the process and finds another runnable process
     - **Success:** Context switch
     - **Failure:** Wait for interrupt (machine instruction)
  3. Device signals it is ready to be accessed
     - Waiting process becomes **runnable**
     - Retry operation from (1)

- **How does the waiting process know to retry the operation from (1) when made runnable?**

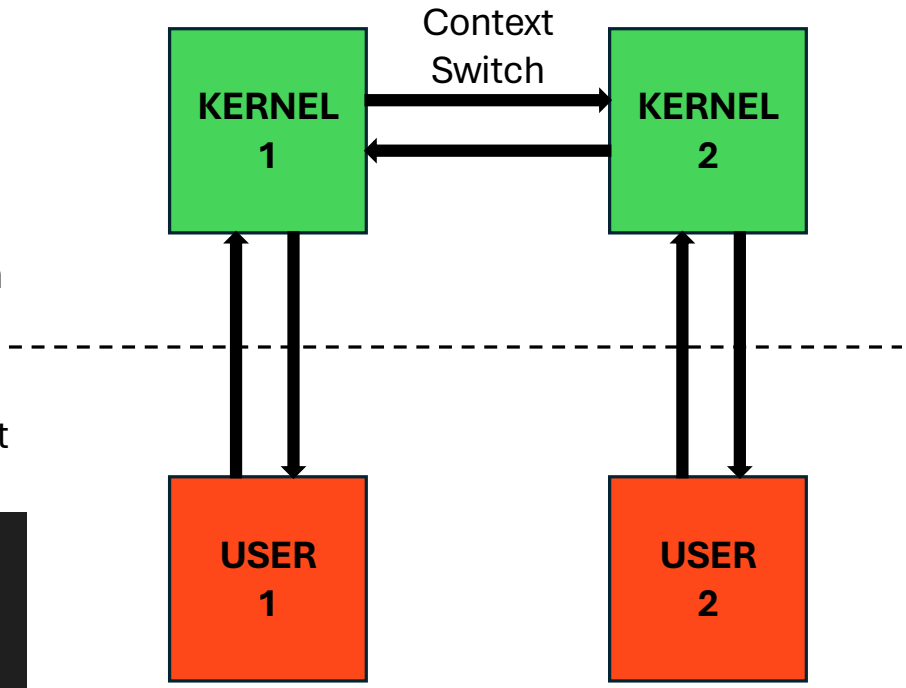


External Interrupts: Classic Approach

Kernel Stack per Process:

- **The OS maintains two states for each process:**
  1. User State: Context of the user executing before the exception/interrupt
  2. Kernel State: Context of the kernel who is in the middle of handling the exception/interrupt **for that specific process**
- Need to retain kernel state of process if a device access fails
  - **On device signal**, processes waiting to interact with device become runnable
  - Context Switch to waiting (now runnable) process, restore all registers from the retained kernel state
  - Process returns to device access and retries request

```
struct process
{
    int pid;
    int status;
    int receiver_pid; /* used when waiting to send a message */
    void *sp, *mepc;  /* process context = stack pointer (sp)
                       * + machine exception program counter (mepc) */
    void *ksp;        /* kernel stack pointer */
    char stack[STACK_SIZE];
};
```



External Interrupts: Context Switching

- On Context Switch:
  1. Our Stack Pointer is currently inside of the **Process-Specific** Kernel Stack
  2. We save the remaining Callee-Saved Registers and Return Address of how the Kernel is currently operating,
     - Allows for us to eventually restore Kernel State and return to the device access
     - No need to save other Caller-Saved Registers, call to ctx_switch() will save them for us on the Process-Specific **Kernel Stack** by calling conventions
  3. Save SP of current top of Kernel Stack in Process' PCB
  4. Switch SP to Kernel we wish to run, Restore Kernel Context

```
ctx_switch:
    addi sp,sp,-64
    sw s0,4(sp)       /* Save callee-saved registers */
    sw s1,8(sp)
    sw s2,12(sp)
    sw s3,16(sp)
    sw s4,20(sp)
    sw s5,24(sp)
    sw s6,28(sp)
    sw s7,32(sp)
    sw s8,36(sp)
    sw s9,40(sp)
    sw s10,44(sp)
    sw s11,48(sp)
    sw ra,52(sp)      /* Save return address */
    sw sp,0(a0)       /* Save the current stack pointer */
    mv sp,a1          /* Switch the stack */
    lw s0,4(sp)       /* Restore callee-saved registers */
    lw s1,8(sp)
    lw s2,12(sp)
    lw s3,16(sp)
    lw s4,20(sp)
    lw s5,24(sp)
    lw s6,28(sp)
    lw s7,32(sp)
    lw s8,36(sp)
    lw s9,40(sp)
    lw s10,44(sp)
    lw s11,48(sp)
    lw ra,52(sp)      /* Restore return address */
    addi sp,sp,64
    ret
```

External Interrupts: Reading Character

```
int read_chars(char *buf, int len)
    for (int i = 0; i < len - 1; i++)
    {
        grass->sys_device(GPID_SHELL, DEV_TTY_READ, &c);
        buf[i] = (char)c;
```

SYSTEM CALL

```
static void proc_device(struct syscall *sc)
    {
        switch (device)
        {
        case DEV_TTY_READ:
            rc = earth->tty_readc((char *)ret_val);
            if (rc < 0)
                earth->tty_push(curr_pid);
            break;
        }

        if (rc < 0)
        {
            proc_set_waiting(curr_pid);
            proc_yield();
        }
    } while (rc < 0);
}
```

CHARACTER READ

FAILURE

FAILURE

SIGNAL

KEYBOARD PRESS

```
void intr_determine()
        break;
    case PLIC_UART0_INTR:
        run_next = earth->tty_pop();
        break;
    }

    if (run_next >= 0)
        proc_set_runnable(run_next);
```

SIGNAL

```
static void proc_yield()
    int mepc = (int)proc_set[proc_curr_idx].mepc;
    asm("csrw mepc, %0" ::"r"(mepc));
    ctx_switch(&proc_set[old_idx].ksp, proc_set[proc_curr_idx].ksp);
}
```

*** idx of next process ***

- Overview of OS capable of running on constrained hardware
- Recap of traditional OS techniques to improve performance
- Why the techniques are undesirable on constrained hardware
- Motivation for multiplexing in constrained environment
- OS Design for Efficient Multiplexing in Constrained Environments

The Classic Approach is Undesirable in Memory Constrained Situations

Memory is Limited:

- ❖EGOS 2000 was built on the SiFive-E CPU for low area and power solutions
- ❖EGOS 2000 Space for Grass Layer (Kernel) limited to **10 KB**
- ❖Grass Layer Memory Details:
  - Start: 0x0800_2800
  - End: 0x0800_5000
  - Highest Address Used: 0x0800_4B70
  - **Amount Left To Use:** 0x0490 = **1168 Bytes**

- ❖Kernel Stack per Process: Memory Calculation
  - **1 KB** Stack Size
  - **16** Process Control Blocks
  - **16 KB** of space used

It is Challenging to Provide Traditional Resource Multiplexing in These Settings

Why is Limited Memory an Issue:

- Many Operating Systems are Generous with memory usage, and thus cannot be used in memory constrained platforms:
  - Traditional OSes support **hundreds of processes**, each with their own Kernel Stack
    - ❖ 1 KB Kernel Stack
    - ❖ 100 Processes
    - ❖ 100 KB just for Kernel Stacks
    - Currently only support **16 Processes** on the ARTY-A7 Board
  - Can use dynamic data structures to manage Processes, Memory, and IPC
    - ❖ Queue to select next runnable Process (Scheduling)
    - ❖ Free List to allocate Pages to Process (Memory)
    - ❖ Message Queue per Process (IPC)

- Embedded Linux, with necessary driver code, is at least 500 KB in size when fully optimized for memory consumption (ex: gcc -Os)

It is Challenging to Provide Traditional Resource Multiplexing in These Settings

Why is Limited Memory an Issue:

- To run on constrained platforms, the OS is required to sacrifice efficiency by limiting the size of process specific data structures
  - **Consequence:** Processes can no longer have their own kernel stacks, implying they cannot block on devices during system calls

- Embedded Linux has noticeably reduced performance with lower RAM, which is undesirable for tail-latency sensitive applications

Evaluating Classic Approach on Real Constrained Hardware

- ARTY-A7 SiFivE-E CPU DESIGN: No more room for Pages past Memory Location 0x0800_8000
  - Hang on final Page Allocation
  - Memory acts as bottleneck, we want our Operating System to run on hardware
  - Need a design shift, can no longer retain state of System Call execution **per process**
- Desire: Device Request & Interrupt Handling **Without Saved State**
  - Would Give:
    - Pros of External Interrupts (no busy waiting)
    - Little Extra Memory Necessary (< 4 KB)
  - Would Lose:
    - Blocking on devices inside of a system call
    - Efficiency, as any data structure optimizations would use too much memory

```
[CRITICAL] Unalloc Frame 241
[CRITICAL] Unalloc Frame 242
[CRITICAL] Unalloc Frame 243
[CRITICAL] Unalloc Frame 244
[CRITICAL] Unalloc Frame 245
[CRITICAL] Unalloc Frame 246
[CRITICAL] Unalloc Frame 247
[CRITICAL] Unalloc Frame 248
[CRITICAL] Unalloc Frame 249
[CRITICAL] Unalloc Frame 250
[CRITICAL] Unalloc Frame 251
[CRITICAL] Unalloc Frame 252
[CRITICAL] Unalloc Frame 253
[CRITICAL] Unalloc Frame 254
[CRITICAL] Unalloc Frame 255
[CRITICAL] Alloc Frame 0
[CRITICAL] Alloc Page No: 8006
[CRITICAL] Alloc Frame 1
[CRITICAL] Alloc Page No: 8007
[CRITICAL] Alloc Frame 2
[CRITICAL] Alloc Page No: 8008
```

OS dies :(

- Overview of OS capable of running on constrained hardware
- Recap of traditional OS techniques to improve performance
- Why the techniques are undesirable on constrained hardware
- Motivation for multiplexing in constrained environments
- OS Design for Efficient Multiplexing in Constrained Environments

Where Constrained Multiplexing is Useful: Sensors on Edge Devices

- **Prior Research Tackled a Similar Problem but for Sensors (Peter Barth; Raphael Barth):**
  - Sensors on edge devices run in constrained environments
    - Space, net bandwidth, energy consumption, costs
  - Sensors are used for different purposes:
    - App 1: Monitoring for dangerous CO2 levels once every 10 seconds
      - High accuracy but requires low data transmission rate
    - App 2: Predictive maintenance by reading once every 100 ms
      - Lower accuracy but requires high data transmission rate

Where Constrained Multiplexing is Useful: Sensors on Edge Devices

- **Prior Research Tackled a Similar Problem but for Sensors (Peter Barth; Raphael Barth):**
  - Giving each application their own edge device is costly, and possibly infeasible due to space and bandwidth constraints
  - Sharing the sensor by running both applications on the same device is integral, but makes sensor configuration management challenging
    - It can be power wasteful to always give the default high accuracy and transmission configuration when, for example, app 2 is suspended
  - The authors provide a virtualized config management interface to apps that meets each application's safety constraints

Why our Research is Novel

- **We are concerned with memory constrained environments, but the motivation is similar**

- Overview of OS capable of running on constrained hardware
- Recap of traditional OS techniques to improve performance
- Why the techniques are undesirable on constrained hardware
- Motivation for multiplexing in constrained environment
- **OS Design for Efficient Multiplexing in Constrained Environments**

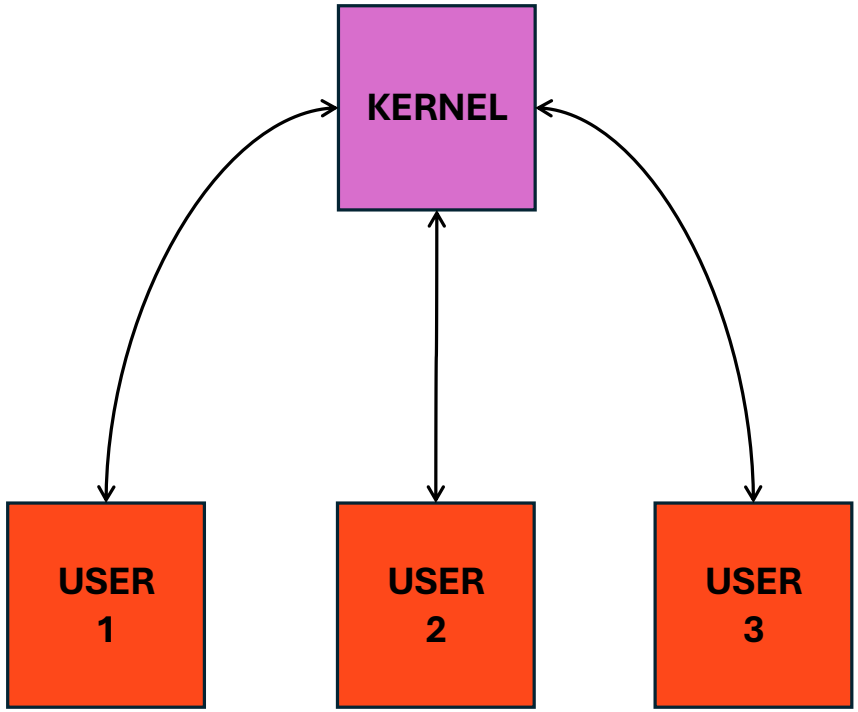
Solution to Memory Constraints

Idempotent (State-less) Kernel

1. **All Processes Share One Kernel Stack**
   - Reserved 8 KB Address Space
   - On Trap Entry, move to **Top of Kernel Stack**, overwriting previous contents
   - Can no longer retain the state of Process-Specific device access on Kernel Stack

2. **Requests are Transactional**
   - **Success:** Fully Completed
   - **Failure:** Nothing Completed
   - Partial Completion implies saving state **per Process**

3. **No More Context Switching**
   - We cannot have any Process-Specific Kernel Context to switch to
   - Yields move directly back to User Space



Idempotent Kernel: Key Concepts (1)

Requesting Mode:

1. On System Call, Process enters Requesting Mode
2. Service Device Specific Request
   - **Success:** Runnable
   - **Failure:** Requesting
   - **Error:** Runnable
     - Make User Process Retry Request

- As Request Servicing is **Transactional**, nothing is completed on failure, retry until success

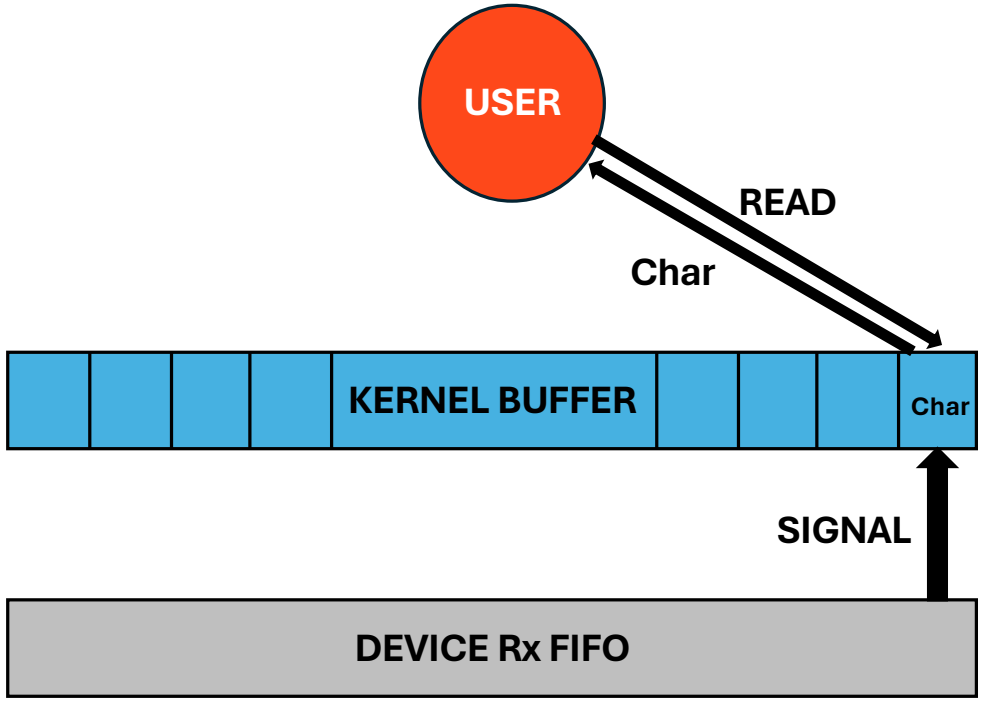
```
static void syscall_handle()
    proc_set_requesting(curr_pid);

    switch (type)
    {
    case SYS_RECV:
        rc = y_recv(sc);
        break;
    case SYS_SEND:
        rc = y_send(sc);
        break;
    case TTY_READ:
        rc = proc_tty_read(sc);
        break;
    case TTY_WRITE:
        rc = proc_tty_write(sc);
        break;
    }

    if (rc == 0)
        proc_set_runnable(curr_pid);
    else if (rc == -1)
        sc->type = type; // Failure, Keep Requesting, Retry Request on External Interrupt
    else
    {
        proc_set_runnable(curr_pid); // Error, Request Will Never Succeed
        sc->retval = -1;
    }
```

Idempotent Kernel: Key Concepts (2)

Buffering: All Requests interact with Kernel Buffers by Transaction

- Writing to Device:
  - Does Kernel Buff have enough space to satisfy request?
    - **Yes:** Write full input into Buff
    - **No:** Write no input into Buff
- Reading from Device:
  - Does Kernel Buff have enough Bytes to satisfy request?
    - **Yes:** Read full output
    - **No:** Read no output
- Abstracts away **device asynchrony** and **irregular I/O formatting**
  - Device can output at **any time**
  - Each device may have different I/O Format
- **External Interrupts handle Buffer to Device Interactions**

```
struct tty_buff
{
    char buf[DEV_BUFF_SIZE];
    int size;
    int head;
    int tail;
};

struct tty_buff tty_read_buf;
struct tty_buff tty_write_buf;
```



Idempotent Kernel: Key Concepts (3)

External Interrupt Handling: On Signal

1. **Handle Device Specific Interrupt**
   - Device Output Queue Filled: Read into Kernel Buffer
   - Device Input Queue Freed: Write from Kernel Buffer
2. **Scan through all Processes**
   - Requesting:
     - We run the process' system call again from the start
   - Possibly Succeed:
     - Kernel Buffers may be filled/freed by External Interrupt
3. **Schedule Next Process**
   - Waiting Processes may have Succeeded
   - If None Runnable: **Wait for Interrupt**, retry from (1)

```c
int external_handle()
    int rc = earth->trap_external(); // Handle Device Specific Request      You,
    int orig_proc_idx = proc_curr_idx;

    /* Keyboard Interrupt */
    if (rc == RET_SPECIAL_CHAR)
        special_handle();

    for (int i = 0; i < MAX_NPROCESS; i++)
    {
        if (proc_set[i].status == PROC_REQUESTING)
        {
            proc_curr_idx = i;
            earth->mmu_switch(proc_set[i].pid);
            syscall_handle(); // Run Requesting Process' Syscall
        }
    }

    proc_curr_idx = orig_proc_idx;
    earth->mmu_switch(curr_pid);
```

```c
void proc_wait()
{
    int mie;
    asm("csrr %0, mie" : "=r"(mie));
    asm("csrw mie, %0" ::"r"(mie & ~(0x88))); // Invert Timer and Software Interrupt Bits

    asm("wfi"); // Interrupt Signal Resumes Execution at PC + 4

    asm("csrw mie, %0" ::"r"(mie | 0x88)); // Enable Timer and Software Interrupt Bits
    external_handle();
}
```

Idempotent Kernel: Key Concepts (4)

Kernel Buffer Bypassing:

- ❖ Writes/Reads to Devices by the Kernel **should not** interact with Buffers
  - **Want Kernel to have Priority Access to Device**
  - Pros:
    - Easier to Debug
    - Simple OS Initialization Process
  - Cons:
    - Busy Waiting for Priority Access

```c
void tty_read_kernel(char *buf, int len)
{
    for (int i = 0; i < len; i++)
    {
        for (c = -1; c == -1; uart_getc(&c))
            ;
        buf[i] = (char)c;
    }
}
```

```c
int tty_write_kernel(char *msg, int len)
{
    int rc;
    for (int i = 0; i < len; i++)
    {
        do
        {
            rc = uart_putc(msg[i]);
        } while (rc == -1);
    }
    return len; // Must Return Length of Message Written to _write()
```

Low Level Details

Idempotent Kernel: Trap Entry/Exit

We only maintain a single Entry/Exit point to/from The Shared Kernel Stack

- Idempotent Workflow: On Exception/Interrupt
  1. Move SP from User Space to single Kernel Stack
     - **How do we obtain User SP again after initially entering Kernel Stack?**
  2. Save **all** General-Purpose User Registers at once, as we have no Kernel State to eventually save
  3. Save User SP and MEPC in Process Control Block
  4. Handle Interrupt/Exception **completely**
     - Complete Everything or Complete Nothing
  5. Switch back to User Space of some Runnable process

- **Where are the User Registers Saved if the Kernel Stack can be overwritten**?

```
void ctx_entry()
{
    int mepc, sp;
    asm("csrr %0, mepc" : "=r"(mepc));
    asm("csrr %0, mscratch" : "=r"(sp));
    proc_set[proc_curr_idx].mepc = (void *)mepc;
    proc_set[proc_curr_idx].sp = (void *)sp;

    /* kernel_entry() is either proc_yield(), proc_syscall(), or proc_external() */
    kernel_entry();

    /* Switch back to the user application stack */
    mepc = (int)proc_set[proc_curr_idx].mepc;
    sp = (int)proc_set[proc_curr_idx].sp;
    asm("csrw mepc, %0" ::"r"(mepc));
    asm("csrw mscratch, %0" ::"r"(sp));

    earth->tty_user_mode();
    ctx_jump();
}
```

Idempotent Kernel: Trap Entry

How to obtain User SP Again:

- Write into Special Mscratch Register
- Read Mscratch to obtain User SP and place into Process' PCB

Where are User Registers Saved:

- Need stateful address space to hold registers for indeterminate amount of time
- **Memory Bottleneck:** The OS saves all user registers on User Stack
  - Pros: Memory Efficient, Stateful
  - Cons: Insecure, possibly malicious
- Use t0 as User SP, then save original value of t0 on User Stack at end

```
.macro TRAP_START
    /* Write User SP into MScratch */
    csrw mscratch, sp
    /* Adjust SP to top of Kernel Stack */
    lui sp, 0x80004
    addi sp, sp, -0x80
    /* Swap t0 and User SP */
    csrrw t0, mscratch, t0
    addi t0, t0, -116
    /* Save RA of Interrupted Procedure */
    sw ra, 108(t0)
    /* Save all Arguments Used in User Level Execution */
    sw a7, 104(t0)
    sw a6, 100(t0)
    sw a5, 96(t0)
    sw a4, 92(t0)
    sw a3, 88(t0)
    sw a2, 84(t0)
    sw a1, 80(t0)
    sw a0, 76(t0)
    /* Save all Temporaries Used in User Level Execution */
    sw t6, 72(t0)
    sw t5, 68(t0)
    sw t4, 64(t0)
    sw t3, 60(t0)
    sw t2, 56(t0)
    sw t1, 52(t0)
    /* Save all Saved Registers for Idempotency */
    sw s11,44(t0)
    sw s10,40(t0)
    sw s9, 36(t0)
    sw s8, 32(t0)
    sw s7, 28(t0)
    sw s6, 24(t0)
    sw s5, 20(t0)
    sw s4, 16(t0)
    sw s3, 12(t0)
    sw s2, 8(t0)
    sw s1, 4(t0)
    sw s0, 0(t0)
    /* Write Original User t0 Value on User Stack */
    csrr t1, mscratch
    sw t1, 48(t0)
    /* Write Updated User SP into MScratch */
    csrw mscratch, t0
.endm
```

System Calls: Initial Problem

Idempotency: How does Kernel use System Call arguments across failures without State?

Solution: User saves System Call arguments in User Space Global Buffer

- **Swapping:** On System Call Request, we swap to Process-Specific Pages that hold Global Buffer

```
struct syscall
{
    enum syscall_type type; /* Type of the system call */
    struct sys_msg msg;     /* Data of the system call */
    int retval;             /* Return value of the system call */
};
```

```
static struct syscall *sc = (struct syscall *)SYSCALL_ARG;
```

System Call: Initial Problems

```
int _write(int file, char *ptr, int len)
{
    if (file != STDOUT_FILENO)
        return -1;

    if (is_kernel)
        return earth->tty_write_kernel(ptr, len);

    if (len > DEV_BUFF_SIZE)
    {       You, last month • Partial Send/Recv Implementation …
        grass->sys_tty_write(ptr, DEV_BUFF_SIZE);
        return DEV_BUFF_SIZE;
    }

    grass->sys_tty_write(ptr, len);
    return len; // Printf() Expects Amount of Input Written as Return Code
}
```

Keyboard Reads/Writes go directly to TTY Device

- ❖ Why it is a Problem:
  1. Non-Blocking: We desire **Blocking**
     - User Processes should only run when request is serviced
  2. Complex Requesting Implementation:
     - Cannot easily retry request in Kernel Space
     - Harder request generalization to **all Devices**
  3. No Device Protection:
     - User Processes directly access devices in User Space
- ❖ Solution: Route Requests through System Call Interface

```
int read_chars(char *buf, int len)
{
    char c;

    for (int i = 0; i < len - 1; i++)
    {
        grass->sys_tty_read(&c);
        buf[i] = (char)c;
```

System Calls: General Approach

```
enum syscall_type
{
    SYS_UNUSED,
    SYS_RECV,
    SYS_SEND,
    TTY_READ,
    TTY_WRITE,
    SYS_NCALLS
};
```

We Maintain 4 System Calls:

- Send: Send Message to another Process
- Recv: Receive Message from another Process
- TTY Read: Read Character from Keyboard
- TTY Write: Write Character to Screen

System Call Workflow:

1. Save Arguments of System Call in SC Buffer
2. Induce a System Call
3. Exception Handler:
   - Move to Requesting Mode
   - Attempt Request
   - On Failure, External Interrupt runs Request again until Success

```
int sys_tty_read(char *c)
{
    sc->type = TTY_READ;
    memcpy(sc->msg.content, &c, sizeof(char *));
    sys_invoke();
    return sc->retval;
}

int sys_tty_write(char *msg, int len)
{
    sc->type = TTY_WRITE;
    memcpy(sc->msg.content, &msg, sizeof(char *));
    memcpy(sc->msg.content + sizeof(char *), &len, sizeof(int));
    sys_invoke();
    return sc->retval;
}
```

System Calls: Kernel General Format

1. Read out arguments from User Space Global Buffer
2. Call Device-Specific Request with Arguments

```c
static int proc_tty_read(struct syscall *sc)
{
    char *c;
    memcpy(&c, sc->msg.content, sizeof(c)); // Read Character into User Space Pointer

    return earth->tty_read(c);
}

static int proc_tty_write(struct syscall *sc)
{
    char *msg;
    int len;

    memcpy(&msg, sc->msg.content, sizeof(msg));
    memcpy(&len, sc->msg.content + sizeof(msg), sizeof(len));
    return earth->tty_write(msg, len);
}
```

System Call: TTY Read

Transactional: Raw Mode

- **No Chars:** Fail
- **Some Chars:** Read one, Success

- **<u>On Failure:</u>**
  - External Interrupt will possibly fill buffer
  - Request is Retried

- **How does UART signal that we can read more?**

```
int tty_read(char *ret_val)
{
    if (rx_size == 0)
        return -1;

    *ret_val = (char)tty_read_buf.buf[rx_head];

    rx_head = (rx_head + 1) % DEV_BUFF_SIZE;
    rx_size--;
    return 0;
}
```

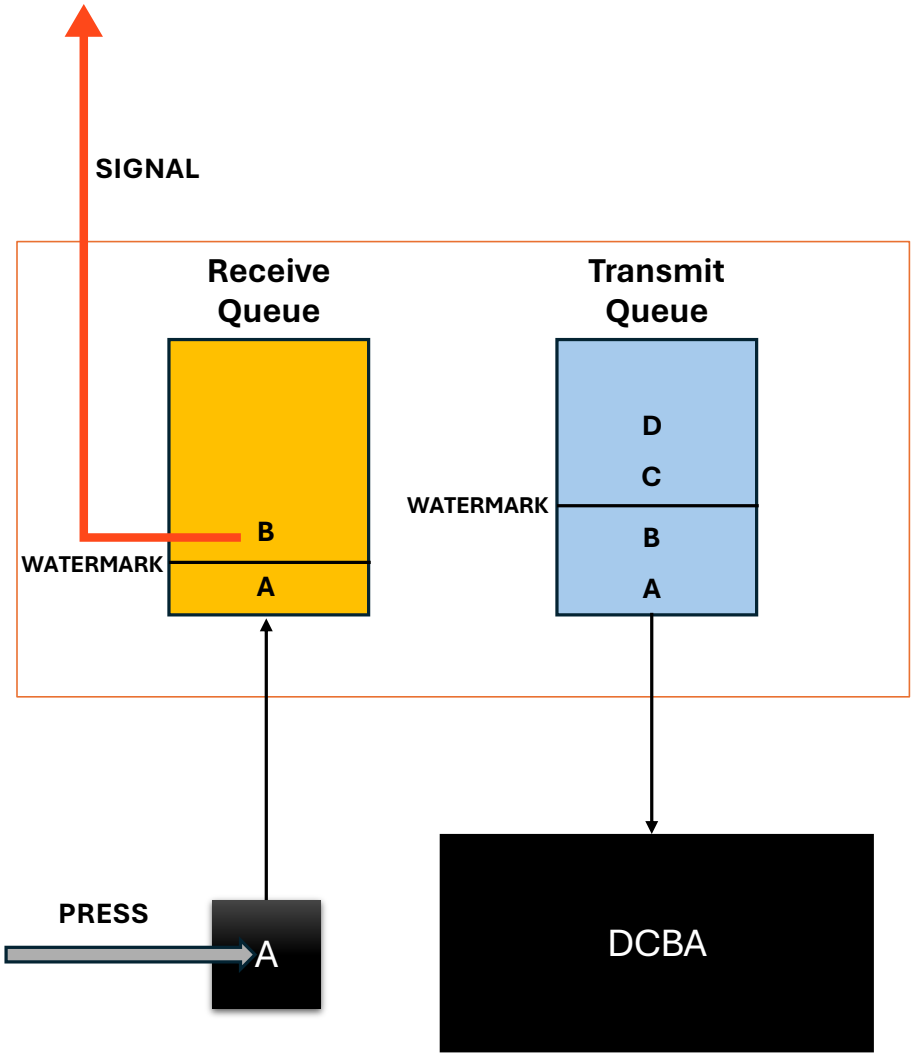
UART Device

Two Async Functions:

1. **Receive** characters from keyboard, held in Rx Queue
2. **Transmit** characters to screen from Tx Queue

- Watermark: Number which decides **when** the CPU is given an Interrupt Signal
  1. Receive Watermark: Signal if Rx FIFO has **more** than Watermark characters
     - Space filled to read more
  2. Transmit Watermark: Signal if Tx FIFO has **less** than Watermark characters
     - Space freed to write more

- **Signal asynchronously handles when we should read/write more characters from/to UART**



System Call: TTY Read UART

```
int tty_read_uart()
    if (uart_getc(&c) == -1)
        return -1;

    do
    {
        tty_read_buf.buf[rx_tail] = (char)c;

        if (rx_size < DEV_BUFF_SIZE)
        {
            rx_tail = (rx_tail + 1) % DEV_BUFF_SIZE;
            rx_size++;
        }

        /* Return to Kernel To Kill Killable Processes */
        if (c == SPECIAL_CTRL_C)
            return RET_SPECIAL_CHAR;

    } while (uart_getc(&c) != -1);
```

UART Device maintains Receive Watermark:

- **Interrupt Signaled:** At **least one** character enqueued onto UART Receive Queue
  1. Read as many characters as possible into Kernel Rx Buffer
  2. **Kernel Retries all Requests:**
     - Kernel Receive Buffer possibly filled
     - TTY Read Requests may Succeed

```
int uart_getc(int *c)
{
    int ch = REGW(UART0_BASE, UART0_RXDATA);
    return *c = (ch & (1 << 31)) ? -1 : (ch & 0xFF); /* Bit 31 Indicates if Empty */
}
```

****

Will Come Back to CTRL-C

****

System Call: TTY Write

To maintain Transactions, we Form 3 Cases:

1. **String Size > Total Buffer Size**: Error
   - Can never fit inside of Buffer
   - All requests **must interact** through Buffer
2. **String Size > Remaining Buff Size:** Failure
   - **External Interrupt** by UART will flush Kernel Buffer
     - Signaled when space is freed on UART Transmit Queue
   - **Retry Request** when remaining size is *possibly* larger
3. **String Size ≤ Remaining Buff Size**: Success
   - Message -> Kernel Buffer -> UART

- **External Interrupt by Transmit Watermark manages writing any remaining Buffer Contents into UART**

```c
int tty_write(char *msg, int len)
{
    if (len > DEV_BUFF_SIZE)
        return -2; // Error, Request will never succeed

    if (len > DEV_BUFF_SIZE - tx_size)
        return -1; // Failure, Retry after External Interrupt

    /* Write Contents into Buffer */
    tty_write_buff(msg, len);
    /* Write Buffer into UART0 */
    tty_write_uart();

    return 0;
}
```

```c
void tty_write_buff(char *msg, int len)
{
    /* Only Called when Entire Input Fits in Buff */
    if (len > DEV_BUFF_SIZE - tx_size)
        return;

    for (int i = 0; i < len; i++)
    {
        tty_write_buf.buf[tx_tail] = msg[i];
        tx_tail = (tx_tail + 1) % DEV_BUFF_SIZE;
    }

    tx_size += len;
}
```

System Call: TTY Write UART Challenges

1. UART Tx Interrupt Pending is Read Only
2. Tx Below Watermark implies Interrupt Pending Enabled
3. UART Transmit Queue Continues to Signal, even with no Processes Writing

❖ **Result:** Cannot leave Interrupts Enabled

- **Need:** Interrupts Enabled Only When Necessary
- **When Necessary:** UART Transmit Full **and** Kernel Buffer is Non-Empty
  - Want to write more, but cannot until Transmit Queue is emptied
  - On Interrupt, we have actual contents to write into freed space

System Call: TTY Write UART Solution

We Desire the Following Invariant:

UART Tx Interrupt Enabled = Write Buffer is Non-Empty

Main Concepts:

1. **UART Tx Queue Not Full:** Interrupts should not be enabled
   - Pending Bit will Stay On
   - Removes Spurious Interrupts
2. **UART Tx FIFO Full:** Interrupts should be enabled
   - On Interrupt, Remaining Buffer Contents Written to UART

- Once Kernel Buffer fully written, disable interrupts

```
void tty_write_uart()
{
    int rc = 0;

    while (tx_size > 0)
    {
        rc = uart_putc((int)tty_write_buf.buf[tx_head]);

        if (rc == -1)
        {
            uart_txen(); // Tx Full and Buffer Non-Empty, Enable Interrupts
            break;
        }

        tx_head = (tx_head + 1) % DEV_BUFF_SIZE;
        tx_size--;
    };

    if (rc == 0)
        uart_txdis(); // Buffer Empty, Disable Interrupts
}
```

```
int uart_putc(int c)
{
    int is_full = (REGW(UART0_BASE, UART0_TXDATA) & (1 << 31));

    if (is_full)
        return -1;

    REGW(UART0_BASE, UART0_TXDATA) = c;
    return 0;
}
```

System Call: Message Sending (Challenges)

- Problem 1: Receiving
  - What if Message is not Sent Yet?
- Problem 2: Sending
  - What if Receiver is not Receiving Yet?

- **Want:** Send message without waiting for receiver
- **Need:** Message to be retained until receiver reads message

- **Solution 1:** Make Inter-Process Communication **Through Buffers**
  - **Easily Generalizes** to our Requesting Mode
  - **Send Oriented:** Make Receiver wait for Sender
  - But Memory is Limited: Cannot give each Process Message Buffer
    - 1 KB Static Message Buffer
    - 16 Processes
    - 16 KB Just to hold **One Message per Process**

System Call: Message Sending (Challenges)

- **Solution 2:** Single Message Buffer at Bottom of Single Kernel Stack
  - **Pros:**
    1. Memory Efficient
    2. Supports Requesting Mode Interactions with Buffer
  - **Cons:**
    1. Satisfies Deadlock Conditions
       - **Mutual Exclusion:** Only One Message can be Held at a Time
       - **Hold and Wait:** Server Processes **all** Wait to Receive/Send after Sending
       - **No Preemption:** Desired Receiver only one who can Clear Message
       - **Circular Wait:** Desired Message Receiver tries to send **before** receiving

```
struct kernel_msg
{
    int in_use;
    int sender;
    int receiver;
    char msg[SYSCALL_MSG_LEN];
};

struct kernel_msg *KERNEL_MSG_BUFF =
    (struct kernel_msg *)(GRASS_STACK_TOP - GRASS_STACK_SIZE + sizeof(struct grass));
```

System Call: Message Sending (Challenges)

Final Challenge: When should Send/Recv Requests run again?

- Request-like Buffer Interaction, but **Not Interrupt Driven**
- Needs to be Scheduled Just Enough for efficiency

❖ **Enough:** After every failed Send/Recv Request?

- Diabolical Case:
  1. Process Spawner(GPID_PROC) waiting for Kill Message
  2. User Process Exits, Succeeds in sending Exit Message
  3. Requests not run again by Success, but **should be** for Process Spawner to wake up

❖ **Simple Solution:** Run right before scheduling next Process

- Both External Interrupts and Scheduler will run **All Requests**
- **Inefficient:**
  - Only run Requesters **waiting for** Device which Interrupted (Optimization)

```
static void proc_yield()
    /* Find the next runnable process */
    int next_status, next_idx = -1;
    external_handle(); // Call External Handler to Run Possible Requesters

    while (next_idx == -1)
    {
        for (int i = 1; i <= MAX_NPROCESS; i++)
        {
            next_status = proc_set[(proc_curr_idx + i) % MAX_NPROCESS].status;
            if (next_status == PROC_READY || next_status == PROC_RUNNING
                || next_status == PROC_RUNNABLE)
            {
                next_idx = (proc_curr_idx + i) % MAX_NPROCESS;
                break;
            }
        }

        if (next_idx == -1)
            proc_wait(); // Wait for Interrupt
```

System Call: Send

Transactional:

1. **Message Buffer Full:**
   - Fail
   - Try Request again when *possibly* Empty

2. **Message Buffer Empty:**
   - Write Message into Buffer
   - Receiver Needs:
     - Who Sent Message
     - Is Message for Them
   - Succeed

```c
static int y_send(struct syscall *sc)
{
    if (KERNEL_MSG_BUFF->in_use == 1)
    {
        return -1;
    }

    KERNEL_MSG_BUFF->in_use = 1;
    KERNEL_MSG_BUFF->sender = curr_pid; // receiver may not know who is sending
    KERNEL_MSG_BUFF->receiver = sc->msg.receiver;

    memcpy(KERNEL_MSG_BUFF->msg, sc->msg.content, sizeof(sc->msg.content));
    return 0;
}
```

System Call: Receive

Transactional:

1. Message Buffer Empty or Request not for our Process:
   - Fail
   - Try Request again when *possibly* Full, or Request is for our Process

2. Message Buffer Full and Request for our Process:
   - Read Message into User Space Destination
   - Free Buffer for another Send

```c
static int y_recv(struct syscall *sc)
{
    /* No Message Available, or not for Current Process */
    if (KERNEL_MSG_BUFF->in_use == 0 || KERNEL_MSG_BUFF->receiver != curr_pid)
        return -1;

    KERNEL_MSG_BUFF->in_use = 0; // Free MSG Buff for Future Sends

    memcpy(sc->msg.content, KERNEL_MSG_BUFF->msg, sizeof(sc->msg.content));
    sc->msg.sender = KERNEL_MSG_BUFF->sender;

    return 0;
}
```

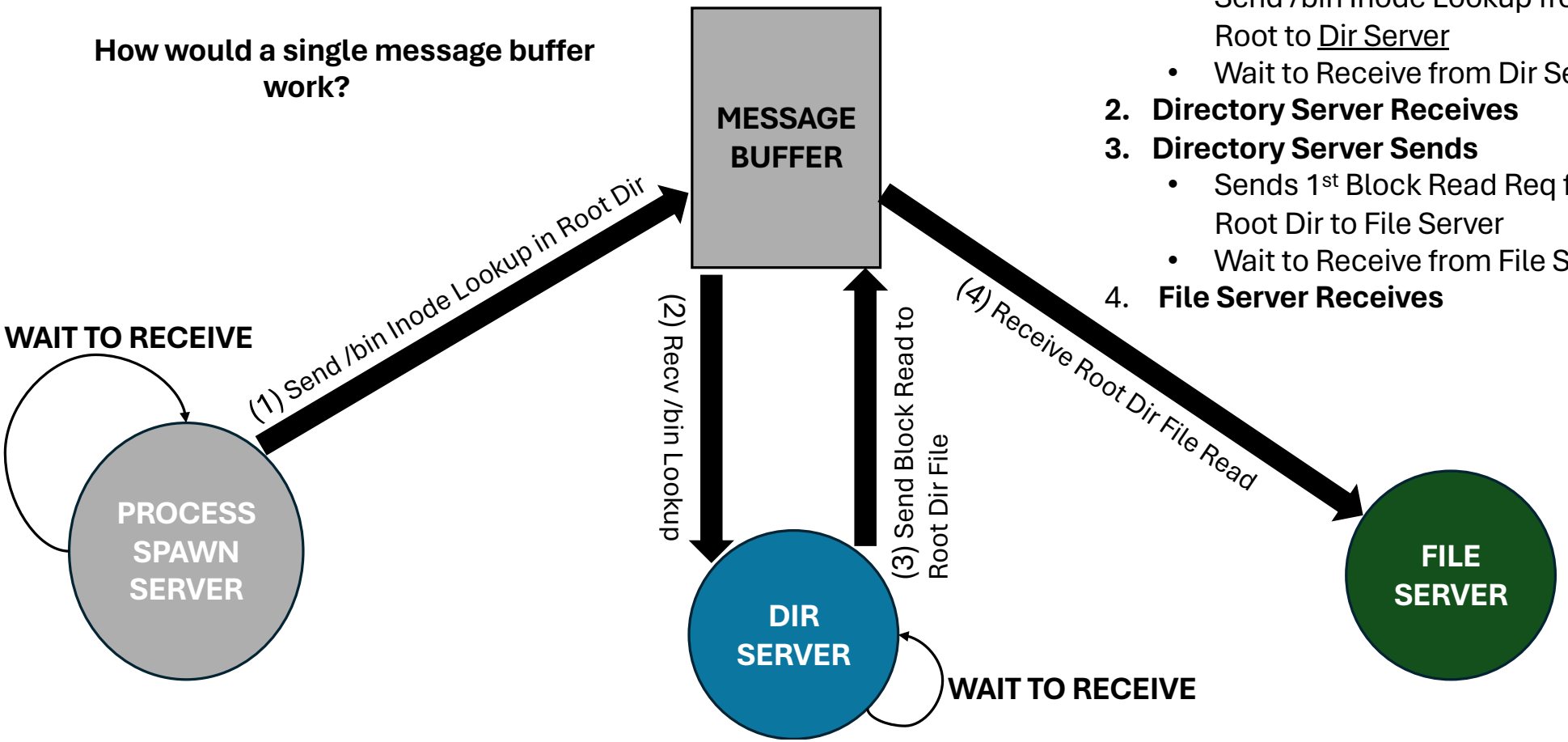
System Call: bin/ Directory Example (Part 1)

Suppose the Process Spawner (GPID PROC) wants to obtain the Inode for the /bin directory, to eventually load the CAT process.

How would a single message buffer work?

Workflow:

1. **Process Spawner Sends**
   - Send /bin Inode Lookup from Root to Dir Server
   - Wait to Receive from Dir Server
2. **Directory Server Receives**
3. **Directory Server Sends**
   - Sends 1st Block Read Req from Root Dir to File Server
   - Wait to Receive from File Server
4. **File Server Receives**



System Call: bin/ Directory Example (Part 2)

We Do Not Deadlock

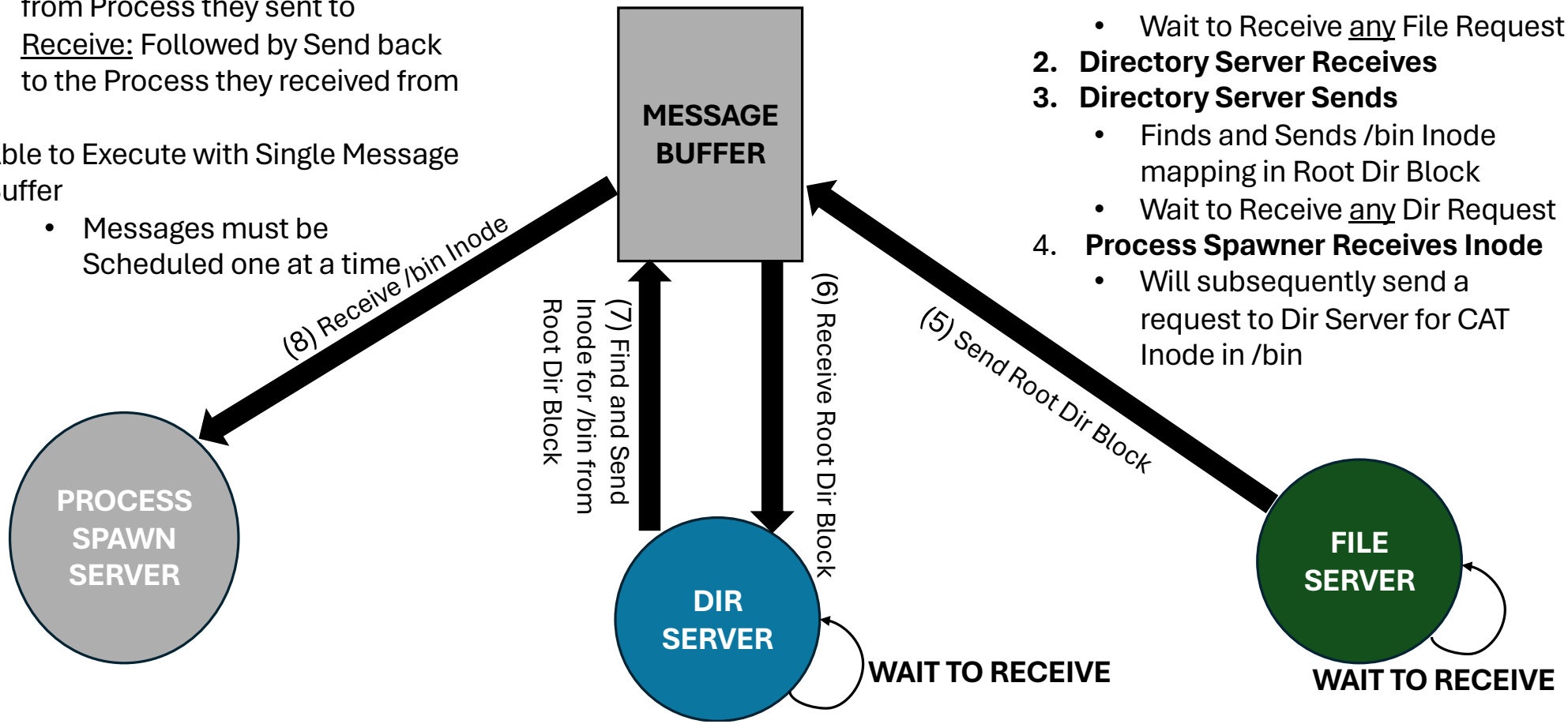
- Send: Followed by Receive back from Process they sent to
- Receive: Followed by Send back to the Process they received from

Able to Execute with Single Message Buffer

- Messages must be Scheduled one at a time

Workflow:

1. **File Server Sends**
   - Send 1st Block of Root Dir
   - Wait to Receive any File Request
2. **Directory Server Receives**
3. **Directory Server Sends**
   - Finds and Sends /bin Inode mapping in Root Dir Block
   - Wait to Receive any Dir Request
4. **Process Spawner Receives Inode**
   - Will subsequently send a request to Dir Server for CAT Inode in /bin



Send/Receive ARTY Deadlock

- **When Running on ARTY Board, we deadlock when attempting to use &**
  - Same issue does not occur on QEMU
  - Likely a Timing Issue with the Single Message Buffer, as the Quanta are different on QEMU and ARTY

```
[INFO] App memory size: 0x00001bb0 bytes
[SUCCESS] Enter kernel process GPID_DIR
[INFO] sys_proc receives: Finish GPID_DIR initialization
[INFO] Load kernel process #4: sys_shell
[INFO] App file size: 0x00000838 bytes
[INFO] App memory size: 0x00001038 bytes
[CRITICAL] Welcome to the egos-2000 shell!
→ /home/yunhao echo hi
hi
→ /home/yunhao cat README
With only 2000 lines of code, egos-2000 implements boot loader, microSD driver, tty driver, memory paging,
 address translation, interrupt handling, process scheduling and messaging, system call, file system, shel
l, 7 user commands and the `mkfs/mkrom` tools.
→ /home/yunhao clock 500 &
[INFO] process 8 running in the background
```

Keyboard Interrupts

Keyboard Interrupts: Control C

1. **Concept:** When CTRL-C is signaled
   - No Killable Processes: Let Shell handle parsing CTRL-C Character
   - Some Killable Processes: Kill all Killable Processes

2. **How do we Kill all Killable Processes?**
   - **Approach 1:**
     - Set MEPC of all Killable Processes to exit() System Call
     - Make all Killable Processes Runnable
   - **Approach 2:**
     - Send **KILLALL** message to Process Spawning Server

```
int tty_read_uart()
    /* Return to Kernel To Kill Killable Processes */
    if (c == SPECIAL_CTRL_C)
        return RET_SPECIAL_CHAR;
```

```
int external_handle()
{
    int rc = earth->trap_external(); // Handle Device Specific Request
    int orig_proc_idx = proc_curr_idx;

    /* Keyboard Interrupt */
    if (rc == RET_SPECIAL_CHAR)
        special_handle();
```

Keyboard Interrupts: Approach 1

- **Set MEPC of every Killable Process to Exit:**
  - Pros:
    - Easy to implement, simply make killable processes Runnable, and they will Send Kill Message to Process Spawn Server
  - Cons:
    - Inefficient:
      - Killed processes take up CPU Time to Exit
    - Deadlocks Easily:
      - Every Killable process attempts to send Kill Message at once to Single Message Buffer
      - May interrupt the Linear Message flow of server processes (/bin Example)

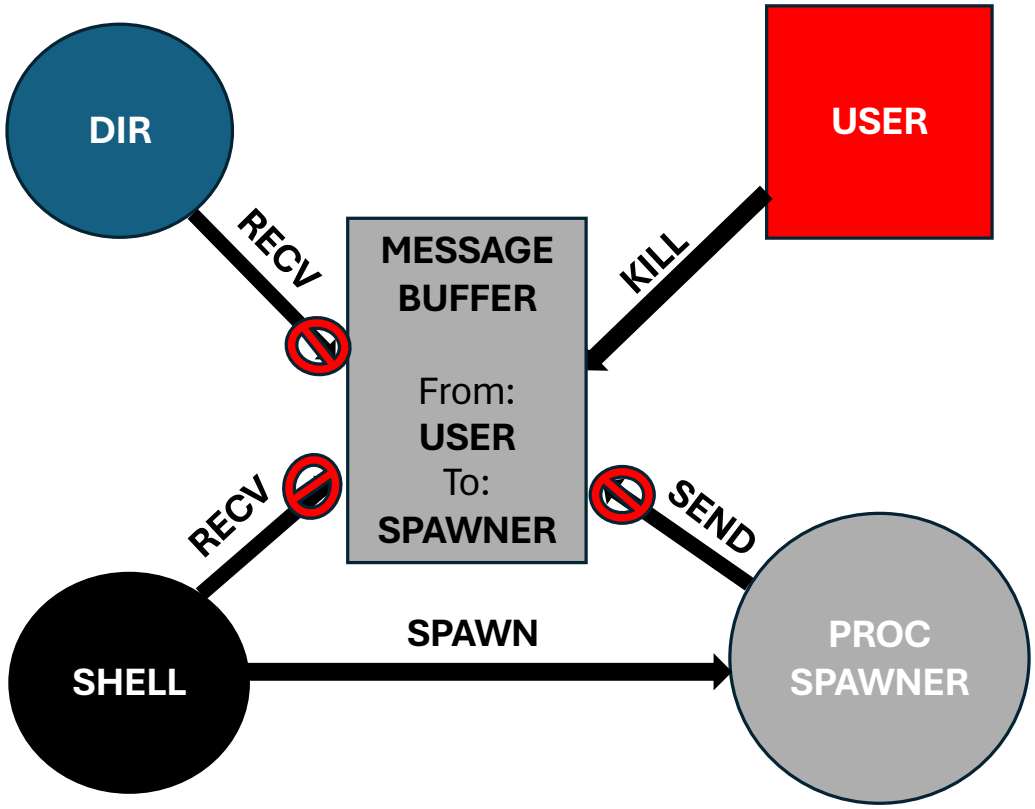
```
int dir_lookup(int dir_ino, char* name) {
    grass->sys_send(GPID_DIR, (void*)&req, sizeof(req));

    grass->sys_recv(&sender, buf, SYSCALL_MSG_LEN);
    if (sender != GPID_DIR) FATAL("dir_lookup: an error occurred, Sender: %d", sender);
    struct dir_reply *reply = (void*)buf;
```

```
clock: tick#153 / #500000
clock: tick#84 / #500000
clock: tick#35 / #500000
clock: tick#154 / #500000
clock: tick#85 / #500000
clock: tick#36 / #500000
[FATAL] dir_lookup: an error occurred, Sender: 21
```

Keyboard Interrupts: Approach 1 Deadlock

- **Setup:**
  1. Shell sends Spawn message to Process Spawner
  2. Process Spawner Recv Message
  3. User Process sends Kill Message to Process Spawner
  4. Process Spawner blocks trying to send Directory Lookup to Dir Server

- **Result:** User is dead, all processes waiting
  - Only one who can receive is Proc Spawner, who is waiting to send
  - Limited Resources



Keyboard Interrupts: Approach 1 Insight

- **Waiting:**
  - We must wait for **all** Server Processes to stop Linear Message Flow
- But how do we guarantee stopping of Linear Message Flow between Servers when multiple Killed Processes attempt to send Kill Message?
  - Example:
    - Shell: Waiting for Keyboard Input
    - CTRL-C: 10 Killable Processes have MEPC Set to Exit()
    - Type CAT README, Shell wakes up, sends Spawn Message to Process Spawner
    - Any one of the 10 Killable Processes Scheduled
    - Devolve to previous Deadlock
  - Problem scales as the number of killable processes grows

- **Guarantee is Non-Trivial:** Only need relatively high chance of success
  - Don't want scaling issue
  - Want: One message to Kill All Processes

```
/* Sending Kill Message while Server Process Active Deadlocks */
if (proc->pid <= GPID_SHELL && proc->status != PROC_UNUSED && proc->status != PROC_REQUESTING)
    return;
```

Keyboard Interrupts: Approach 2 Challenges

- **Send KILLALL Message to Process Spawner:**
  - Problems:
    1. **Which** Killable Process sends the KILLALL Message?
    2. **How** do we prevent the KILLALL Message Sender's execution after Sending?
    3. **What** do we do with the remaining Killable Processes?

Keyboard Interrupts: Approach 2 Concepts

- **Solution to 1:** First Killable Process we find
  - **How** do we make them send the KILLALL Message?
    1. Force Process to make System Call to Send KILLALL
    2. Set Process to Requesting Mode, Message will eventually be sent
- After finishing request, our generalized format makes process runnable
- How do we prevent normal execution, when they should be dead?

```
/* Must be a Killable Process, force to send KILLALL Message */
earth->mmu_switch(proc->pid);

struct proc_request req;
req.type = PROC_KILLALL;
memcpy(sc->msg.content, &req, sizeof(req));
sc->type = SYS_SEND;

proc_set_requesting(proc->pid);
```

Keyboard Interrupts: Approach 2 Concepts

❖ **Solution to 2:** Set MEPC to Idle Loop

- What do we do with the remaining Processes?

```
proc->mepc = proc_idle; // Once Finished Sending Message, make Idle
killall_sent++;
```

```
void proc_idle()
{
    while (1)
        ;
}
```

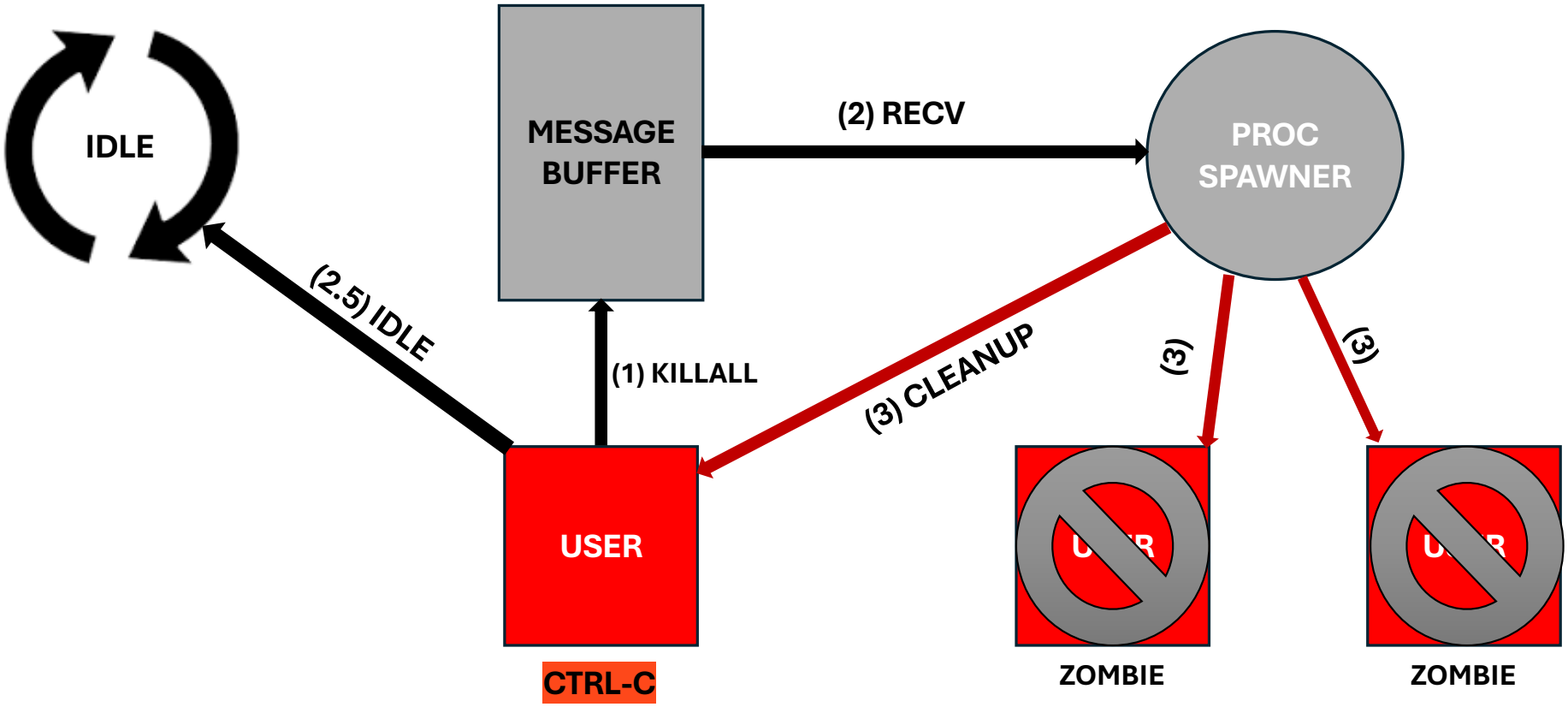
Keyboard Interrupts: Approach 2 Concepts

❖ **Solution to 3:** Set Processes to **ZOMBIE mode**

- Never Scheduled
- Eventually cleaned up by Process Spawner

```
/* Set Remaining Killable Processes as Zombies */
if (killall_sent)
{
    proc_set_zombie(proc->pid);
    continue;
}
```

Keyboard Interrupts: Conceptual Diagram



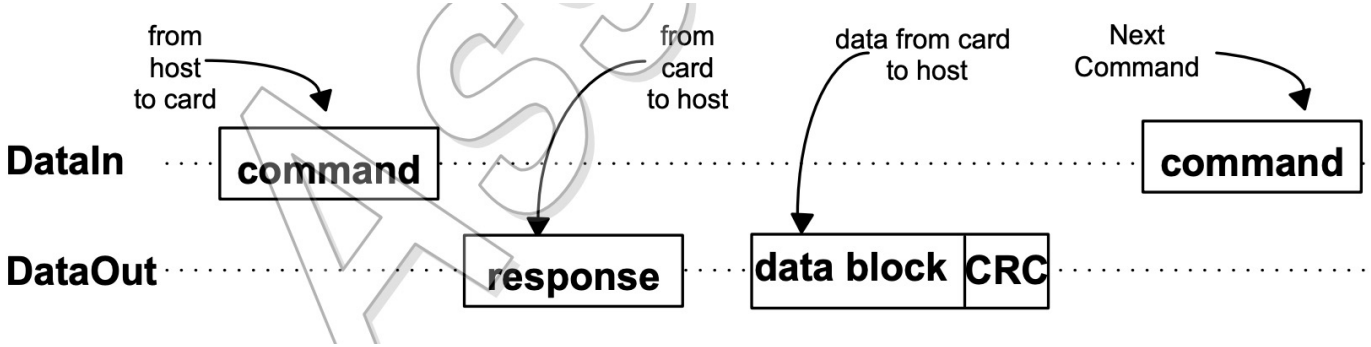
Keyboard Interrupts: Possible Extensions

Possibly Preventing Deadlock:

- If Shell **Not** Waiting to be Woken Up by Process Spawner, we do not allow for Killable Processes to be Killed
    1. Problem: Receiving and Keyboard Reading both use Requesting Mode
        - Need fine-grained details on what our Request is for
        - Fine-Grained: Problems
            1. Greater use of Limited Memory
            2. Less generalization to single Requesting Mode

- Would prevent diabolical spawning case

Source: Part 1 Physical Layer Simplified Specification, Ver 9.10, SD Association



Extensions: Disk Read

- **Must interact with SD Card using Serial Peripheral Interface (SPI)**
  - **General Format:**
    1. <u>SD Read Workflow Format:</u>
       - Send 6 Byte Command to Read Block
       - **SD Card Sends:**
         - Response signaling Request was parsed
         - Start Token of Block
         - Entire Block
         - 2 Byte Checksum

Extensions: Disk Read Problems

SPI Interface Key Concepts:

1. Synchronous: SPI has its **own clock** to serialize sending of data between the SD Card and SPI Queues
2. Inducing Clock: SPI **must send a data byte** to the SD Card to induce (8) clock ticks for a byte to be received by the SD Card
3. Exchanging Bytes: Every send of a data byte to SD Card is correspondingly exchanged with a data byte from the SD Card
4. Identical Rx and Tx to UART

❖ SD Response

- **R1:** Indicates if Command was Successful (Response to Command)
- **0xFF:** Currently Busy with Command
- **0xFE**: Ready to Send Data Block (Start Token)
- **Block Data**

Extensions: Disk Read System Call

- **Transactional:** Must maintain buffers that act as interface to receive **entire block** at once
- **Command Buffer:**
  - Holds 6 Byte Command and remaining 516 Dummy Bytes (0xFF) to induce Clock Signal for each needed SD Card Response
- **Receive Buffer:**
  - Holds the output of the SPI Receive Queue filled by SD Card
  - Once filled, we then **allow** for a Block Read to succeed
- **Need Invariant:** Can only receive bytes after sending
  - ❖ Num Received Bytes ≤ Num Sent Bytes

```
#define READ_SIZE 6 + 1 + 1 + BLOCK_SIZE + 2
```

```
struct disk_buff
{
  char buff[READ_SIZE];
  int idx;
};
```

```
struct disk_msg
{
  int in_use;
  int block_no;
  struct disk_buff command;
  struct disk_buff recv;
};
```

Extensions: Disk Read System Call

- **Transactional:**
  - If DISK_MSG Not In Use:
    - Set in_use, block_no to request parameters
    - Write block_no into command buffer (part of SD Read CMD)
    - Send as much of Command Buffer as Possible
    - Fail
  - If DISK_MSG In Use:
    - If Entire Block Read Out **and** DISK_MSG Holds desired Block No:
      - Read Block to User Space of Process
      - Reset DISK_MSG Buffer
      - Succeed
    - Otherwise: Fail
- **External Interrupts for Rx and Tx will drive Sending the entire Command Buffer and Filling the entire Receive Buffer**

Extensions: Disk Read Concept