# Part 1: Minix Kernel

Minix adopts a **microkernel architecture**, where the kernel is intentionally kept minimal and almost ascetically lean. Instead of embedding device drivers, file systems, and process management logic directly inside the kernel, Minix forces these components to exist as **separate processes in user space**. The result is a system in which **interactions between OS components occur strictly through message passing**. Reliability is achieved by *minimizing what is allowed to run in privileged mode*.

At the heart of this communication model are three core kernel mechanisms:

1. **`mini_send()`** – delivers a message from one process to another.
2. **`mini_receive()`** – waits for and retrieves an incoming message.
3. **`pick_proc()`** – selects the next runnable process from the ready queues and schedules it on the CPU.

Together, these mechanisms form the backbone of process coordination and CPU time allocation in Minix.

## 1. Message Passing: `mini_send()` and `mini_receive()`

Instead of shared-memory synchronization or direct system-call handling, Minix uses **synchronous message passing**.
When a process sends a message using `mini_send()`, one of two outcomes occurs:

- If the receiving process is already waiting for this message, it is delivered immediately.
- Otherwise, the sender is **blocked** until the receiver performs `mini_receive()`.

Similarly, `mini_receive()` blocks a process until a corresponding message is available. This design ensures **tight synchronization without race conditions**. In effect:

```
Sender Process  → mini_send() → [ Kernel Mediates ] → mini_receive() → Receiver Process
```

- No shared memory.
- No direct access.
- The kernel is a tool for fair share of memory and process order.

## 2. Process Scheduling: `pick_proc()`

After any send/receive operation changes the run state of processes, the kernel must decide **which process to run next**. This responsibility falls on `pick_proc()`.

`pick_proc()` iterates through **priority-based ready queues**, from highest to lowest priority, selecting the first available runnable process. This approach ensures that:

- System tasks and servers receive CPU time before user programs.
- Blocking due to message synchronization naturally influences scheduling order.
- The kernel remains extremely compact and deterministic.

In short: **Communication (send/receive) changes who is ready, and `pick_proc()` decides who gets the CPU next.**

## 3. Detailed Behavior of `mini_send()`, `mini_receive()`, and `pick_proc()`

| Field | Meaning |
|---|---|
| `p_rts_flags` | Process state (e.g., RUNNING, SENDING, RECEIVING) |
| `p_sendto` | PID of the process it is waiting to send to |
| `p_getfrom` | PID of the process it expects to receive from |
| `p_messbuf` | Pointer to message buffer |
| `p_priority` | Scheduler priority queue |

A process is runnable when **none** of the `p_rts_flags` bits are set.

## 4. `mini_send()` — Synchronous Send

When a process calls mini_send():

1. Kernel checks whether the receiver is waiting (p_rts_flags & RECEIVING).
2. If yes → message is copied immediately → receiver unblocks.
3. If not → sender is blocked and placed onto a send queue.

The key idea: The sender cannot continue until the receiver handles the message. This enforces synchronization and avoids race conditions.

### 5. `mini_receive()` — Blocking Receive

When a process calls mini_receive():

1. Kernel checks if there is any pending message already waiting.
2. If yes → message is copied → sender unblocks.
3. If no → receiver blocks until someone sends to it.

### 6. Scheduling With `pick_proc()`

Whenever send/receive operations change who is blocked or runnable, the scheduler is invoked:

pick_proc() → selects first runnable process from highest-priority queue

This is why when the sender blocks, the CPU immediately switches to the receiver — even without a timer interrupt.

### 7. Message Flow Example

Sender - - mini_send() {blocked, waits} - - → Kernel - - mini_receive(){message copied} - - → Receiver - (reply)- → Kernel - - receiver reply {unblocked, continues} → Sender

# Part 2: Simple Application

### 1. Sender: sender.c

```
#include <minix/ipc.h>
#include <stdio.h>

int main() {
    message m;
    m.m1_i1 = 100;

    printf("[SENDER] Sending message...\n");
    send(1, &m);   // Send to receiver (endpoint = 1)
    printf("[SENDER] Waiting for reply...\n");
    receive(1, &m);

    printf("[SENDER] Received reply: %d\n", m.m1_i1);
    return 0;
```

```
}
```

## 2. Receiver:receiver.c

```
#include <minix/ipc.h>
#include <stdio.h>

int main() {
    message m;

    printf("[RECEIVER] Waiting...\n");
    receive(0, &m);    // Receive from ANY
    printf("[RECEIVER] Got: %d\n", m.m1_i1);

    m.m1_i1 += 1;      // Modify message
    send(m.m_source, &m);

    printf("[RECEIVER] Reply sent.\n");
    return 0;
}
```

## 3. Compile & run:

```
cc -o sender sender.c
cc -o receiver receiver.c
```

Open two terminals. Terminal 1: type ./receiver  Terminal 2: type ./sender

| Event | Kernel Action | Scheduling Result |
|---|---|---|
| Sender calls `send()` | `mini_send()` blocks it | `pick_proc()` switches to receiver |
| Receiver calls `receive()` | message delivered | receiver runs |
| Receiver sends reply | sender is unblocked | `pick_proc()` returns to sender |

Call Chain:

- Your code:  send()
- Library:  _send() → system call wrapper
- Trap:  int 0x80 / sys_call()
- Kernel:  do_ipc() → mini_send() / mini_receive()
- Scheduler:  pick_proc()

# Part 3: Zombie and Orphan Processes

*Process Creation and Termination in Unix (WSL Demonstration)*

While Minix demonstrates **inter-process communication and scheduling** using message passing, modern Unix/Linux systems (including WSL) illustrate process state transitions particularly well.

Here, we focus on **how processes are created, how they terminate, and what happens when parent-child relationships break.**

The key function is:

pid_t fork(void);

`fork()` creates a **new child process** by duplicating the parent's address space.

Both processes continue execution.

## 1. Orphan Process

An **orphan process** is created when a **parent process exits before its child.**
In Unix design, every orphaned process is **automatically adopted by `init` (PID 1)**.

This provides:

- **Consistent re-parenting**
- **Guaranteed cleanup**
- **No uncontrolled or abandoned processes**

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid > 0) {
        printf("Parent (%d) exiting.\n", getpid());
        return 0;            // parent exits
    } else {
        sleep(5);            // child continues running
        printf("Child (%d) adopted by %d.\n", getpid(), getppid());
    }
}
```

**Run in Wsl:**

gcc orphan.c -o orphan
./orphan

# 2. Zombie Process

A **zombie process** (defunct process) occurs when a **child exits**, but its **parent does not call `wait()`** to read its exit status.

- The child has finished,
- But its entry in the process table remains,
- Waiting for the parent to acknowledge it.

This is intentional — it allows the parent to **retrieve the exit code.**

### Example: Zombie Process

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid = fork();

    if (pid > 0) {
        printf("Parent (%d) sleeping. Child will become zombie.\n", getpid());
        sleep(30);   // parent does NOT call wait()
    } else {
        printf("Child (%d) exiting now.\n", getpid());
        return 0;    // child terminates → becomes zombie
    }
}
```

**Run:**

gcc zombie.c -o zombie

./zombie & ps -l

# Differences:

| Concept | Minix | WSL (Linux) |
|---|---|---|
| Process creation behavior | Simpler Process Manager | Full `fork()` duplication semantics |
| Inter-process communication | **Message Passing (`mini_send`, `mini_receive`)** | Signals, pipes, shared memory, etc. |
| Scheduling | Priority queues via `pick_proc()` | CFS / priority / time-slice |
| Orphans | Re-parented by PM | Re-parented by `init (PID 1)` |
| Zombies | May not persist | **Persist until `wait()` is called** |

# Part 4:

Fork counting problems:

## 1. How many forks

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    printf("A\n");
    fork();
    printf("B\n");
    if (fork() == 0) {
        fork();
        printf("C\n");
    } else {
        fork();
        printf("D\n");
    }
    return 0;
}
```

a. How many processes are created in total?
b. How many times are **A**, **B**, **C**, and **D** printed?
c. Draw the process tree.

## 2. Fork problem on demand:

Write a program that divides an array among four child prcesses, each summing a segment. The parent collects all results and prints the total sum.

# Assignment (2 weeks):

1. You must apply these problems. Show your work on a screen capture video and upload it. (There are many free tools)
2. Your problems:
    a. sender&receiver on minix
    b. Orphan & Zombie process application on wsl
    c. 2 fork problems on the wsl / minix / ubuntu any platform you like
3. In screen caps, for all problems: we will see your namelastname as user, computer name, login name. Eg. manolyaatalay (no caps). Each problem should be shown in a single take.
4. If you fail to show your login name at all screen times, your assignment will be calculated as 0.
5. You will be verbally explaining your work during the video. If it is not in english, you'll be deduced 5 points.
6. It is an individual work. There is no excuse of "together made, one pc available".
7. In the beginning of video introduce yourself. If not, 10 points will be deduced.
8. Every late day will reduce 15 points from your complete work.
9. You do not need additional report.
10. Good luck