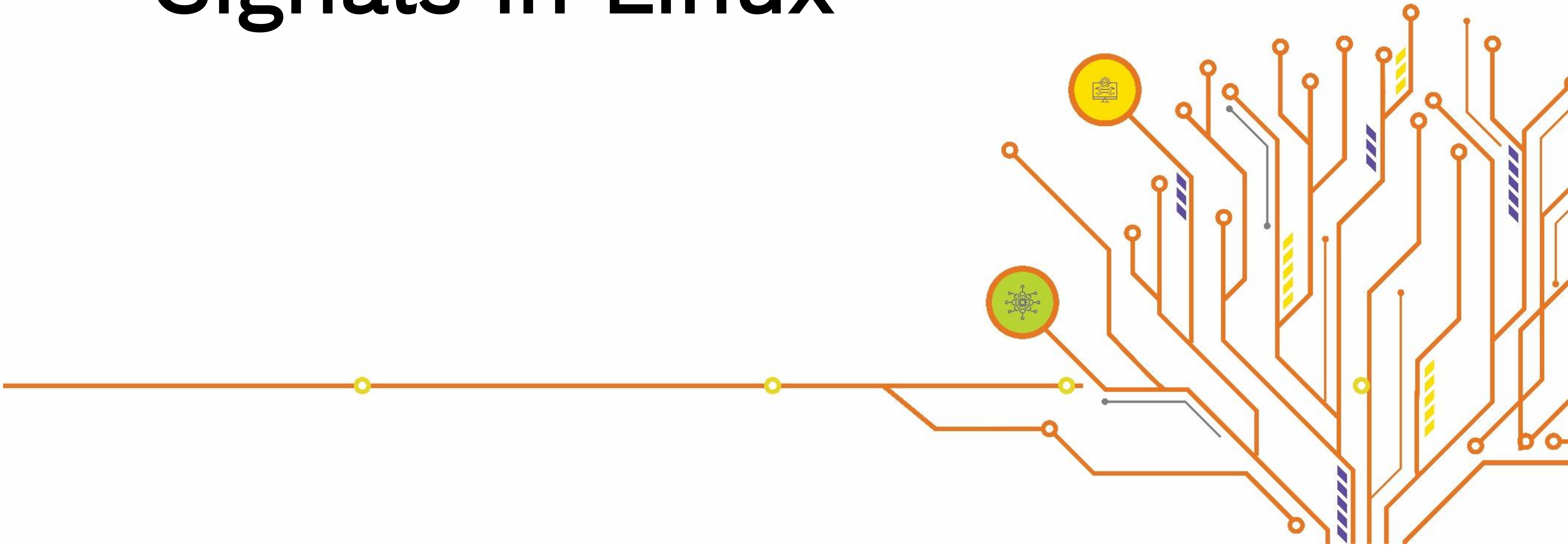
Signals in Linux



Signal

Signals are software interrupts delivered to a process

- They notify a process that some event occurred (e.g., segmentation fault, child exit, timer expiry, CTRL+C, etc.).
- Think of signals as asynchronous notifications sent to a process.
- #include <signal.h>
- Each signal in Linux are identified by unique number assigned to them
- Used for:
  - Process control
  - Exception handling
  - Inter-process communication (IPC – limited but useful)
  - Reporting asynchronous events (division by zero, segmentation fault, timer expiry)

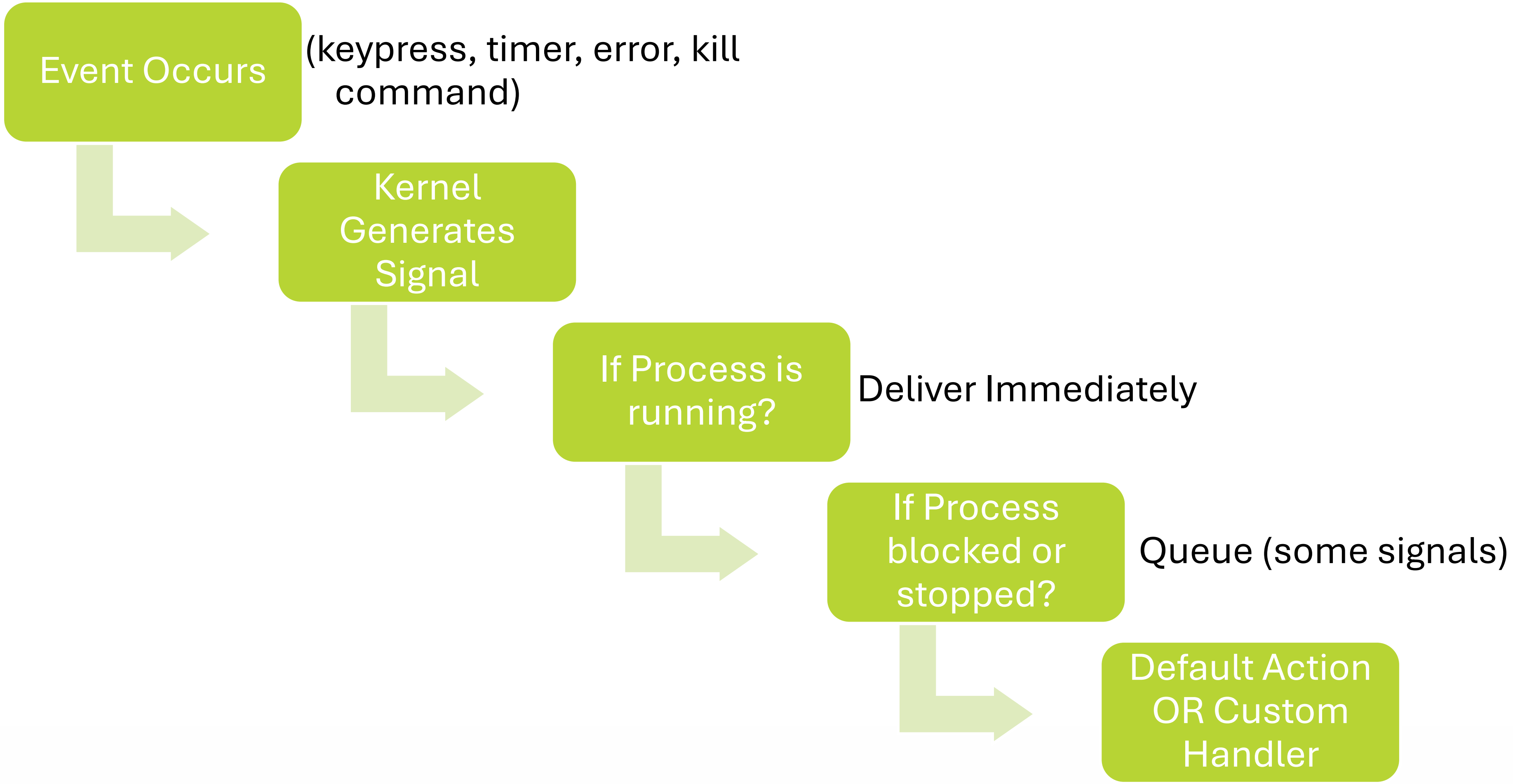
Signal Handling

- Kernel can perform one of the 3 option depending upon what prcess has asked
  - Ignore the signal
  - Catch and handle the signal – Kernel will suspend execution of the process's current code and jumps to a signal handler and the returns to main
  - Perform default action – if there is no user defined signal handler, then default action assigned to that sigal is performed

Common Linux Signals

Signal	Number	Default Action	Meaning
SIGINT	2	Terminate	Ctrl+C
SIGKILL	9	Terminate (cannot be caught/blocked)	Force kill
SIGTERM	15	Terminate	Graceful termination
SIGQUIT	3	Core dump	Ctrl+\
SIGSEGV	11	Core dump	Invalid memory access
SIGABRT	6	Core dump	Abort() called
SIGCHLD	17	Ignore	Child stopped or terminated
SIGSTOP	19	Stop (cannot be caught)	Pause process
SIGCONT	18	Continue	Resume process
SIGALRM	14	Terminate	Alarm timers

Signal Life Cycle



Event Occurs

(keypress, timer, error, kill command)

Kernel Generates Signal

If Process is running?

Deliver Immediately

If Process blocked or stopped?

Queue (some signals)

Default Action OR Custom Handler

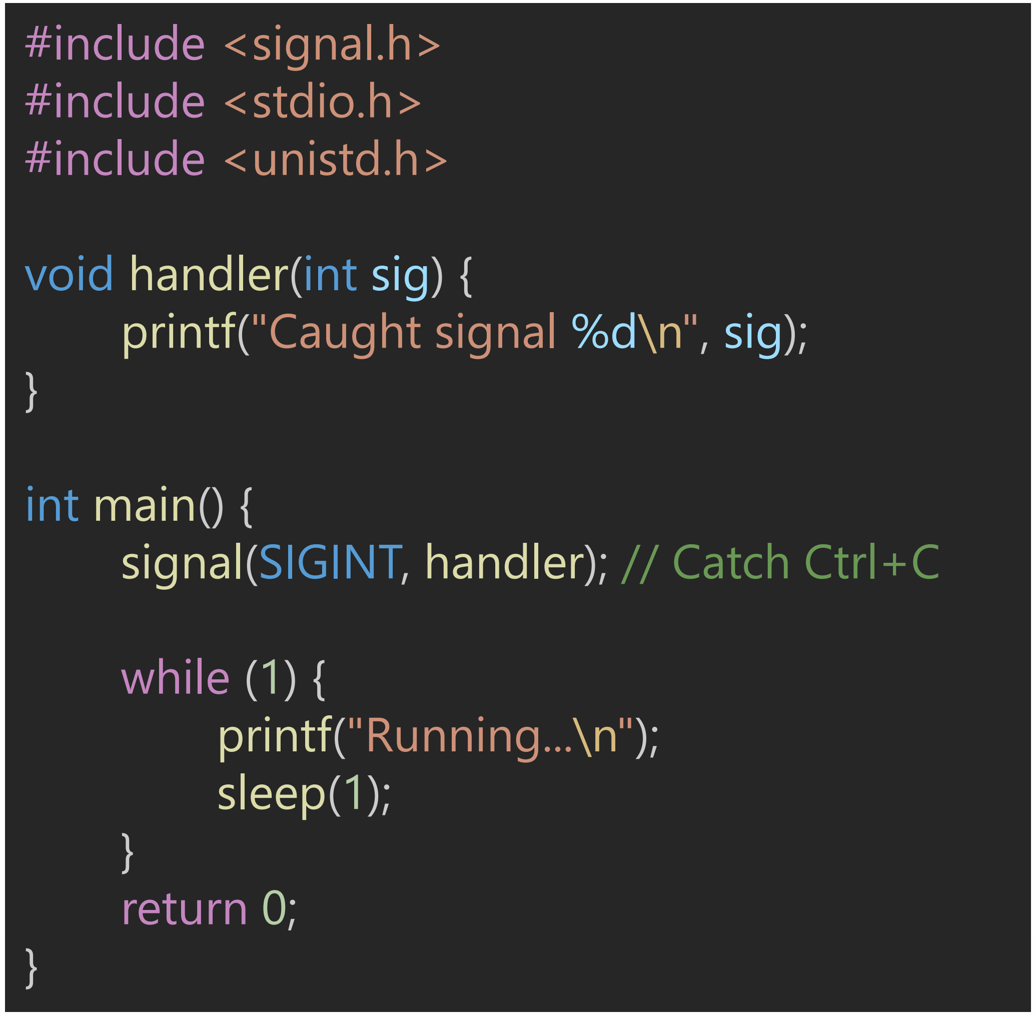
System Calls - Sending Signals

Syntax	Parameters	Return & Errors
int kill(pid_t pid, int sig)	pid: target process sig: signal number	0 on success -1 on error (ESRCH, EPERM)
int raise(int sig)	Sends signal to itself	0 or -1
int tgkill(tgid, tid, sig)	For threads	0 or -1

System Calls - Installing Signal Handlers

Syntax	Parameters	Return & Errors
sighandler_t signal(int sig, sighandler_t func)	sig: signal func: handler function	Old handler or SIG_ERR
int sigaction(int sig, const struct sigaction *act, struct sigaction *old)	act: new actionold: store previous	0 or -1

Catching ctrl+c Signal



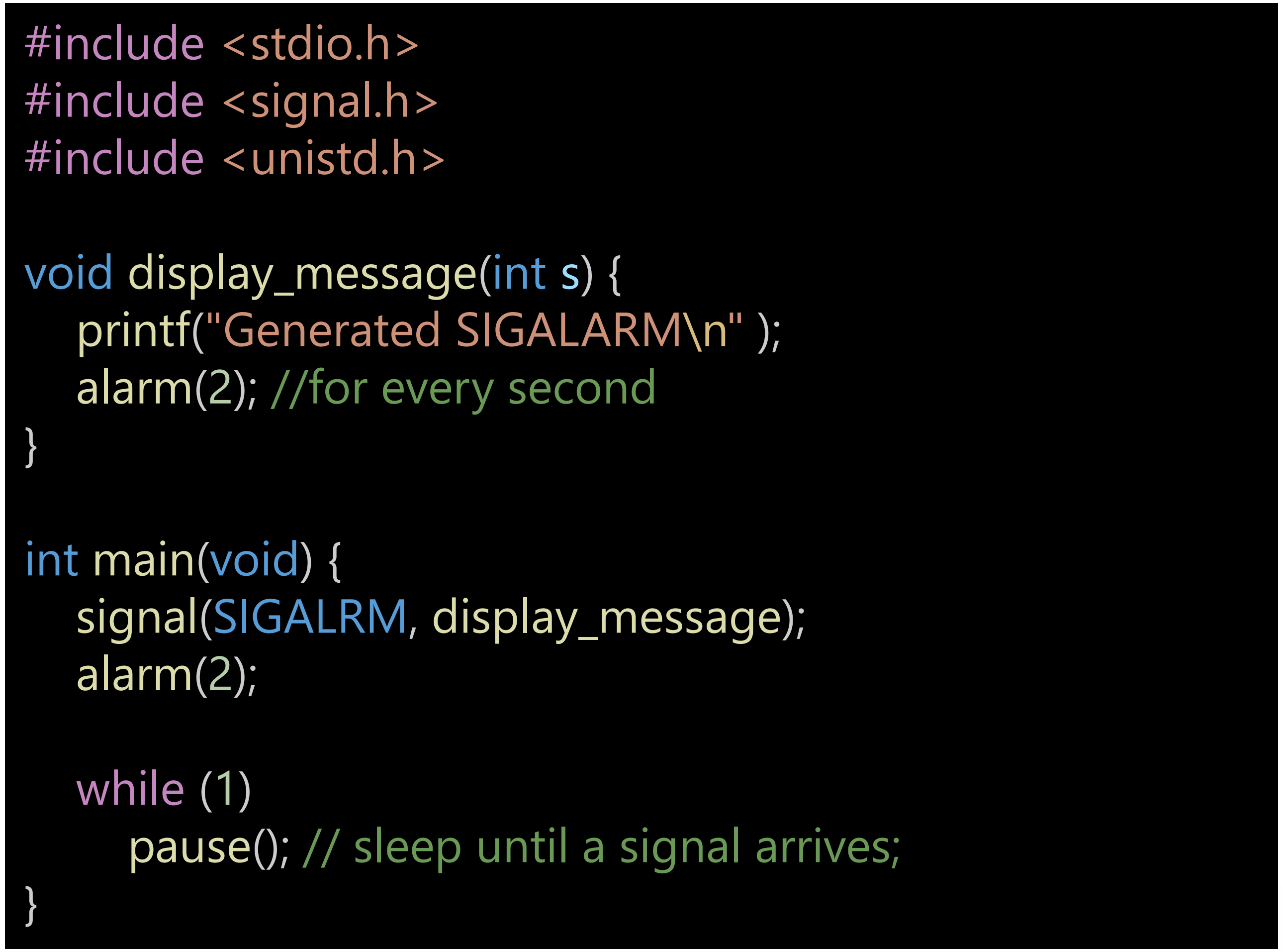
```c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void handler(int sig) {
	printf("Caught signal %d\n", sig);
}

int main() {
	signal(SIGINT, handler); // Catch Ctrl+C

	while (1) {
		printf("Running...\n");
		sleep(1);
	}
	return 0;
}
```

Catching Alarm Signal



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

void display_message(int s) {
    printf("Generated SIGALARM\n" );
    alarm(2); //for every second
}

int main(void) {
    signal(SIGALRM, display_message);
    alarm(2);

    while (1)
        pause(); // sleep until a signal arrives;
}
```

User-Defined Signals

Signal	Number	Default Action	Usage
SIGUSR1	Typically 10	Terminate	Application-defined event
SIGUSR2	Typically 12	Terminate	Application-defined event

User Signal: Parent Receiving Signal from Child

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

void handle_signal(int sig) {
    printf("Parent received signal: %d from child\n",
      sig);
}

int main() {
    signal(SIGUSR1, handle_signal);
    // Register signal handler
    pid_t pid = fork();
```

```
        if (pid < 0) {perror("Fork failed"); exit(1);}

        if (pid == 0) { // Child process
                sleep(2); // Simulate some work
                kill(getppid(), SIGUSR1); // Send signal to
                parent
                exit(0);
        } else { // Parent process
                printf("Parent waiting for signal from
                child...\n");
                pause(); // Wait for signal
                printf("Parent exiting.\n");
        }
        return 0;
}
```

Scenario: Log Monitoring System Using SIGUSR1 Between Unrelated Processes

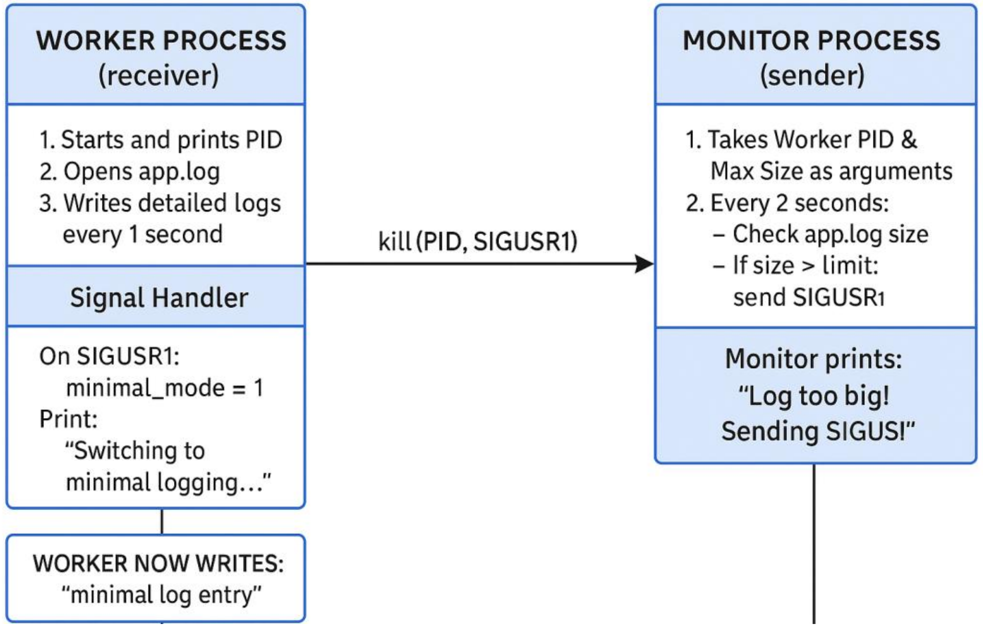
Context:

You are building a simple software setup with two independent processes:

Worker Process: Continuously generates log entries in a file app.log.

Monitor Process: Periodically checks the size of app.log. When the file grows beyond a limit (given by user), the Monitor sends SIGUSR1 to the Worker.

When the Worker receives SIGUSR1, it must switch from detailed logging to minimal logging.



WORKER PROCESS
(receiver)

1. Starts and prints PID
2. Opens app.log
3. Writes detailed logs
   every 1 second

Signal Handler

On SIGUSR1:
  minimal_mode = 1
Print:
  “Switching to
  minimal logging…”

WORKER NOW WRITES:
“minimal log entry”

kill (PID, SIGUSR1)

MONITOR PROCESS
(sender)

1. Takes Worker PID &
   Max Size as arguments
2. Every 2 seconds:
   – Check app.log size
   – If size > limit:
     send SIGUSR1

Monitor prints:
“Log too big!
Sending SIGUS!”

PART 1 — Implement the Worker Process

Create a C file named worker.c

- Print own PID on startup
- Open a file app.log in append mode
- Install a signal handler for SIGUSR1
- Maintain a global flag minimal_mode = 0
- Inside the handler, set minimal_mode = 1
- Keep writing logs every 1 second:
    - If minimal_mode == 0: write "detailed log entry..."
    - Else: write "minimal log entry"

PART 2 — Implement the Monitor Process

Create a C file named monitor.c

- Take two command-line arguments:
  - PID of worker
  - Maximum allowed file size in bytes
- Every 2 seconds:
  - Check size of app.log using stat()
  - If file size > limit → print a message and send SIGUSR1 to worker using kill(pid, SIGUSR1)