# Tutorial 3: Operating Systems

Yonghao Lee

April 19, 2025

## Introduction

This tutorial explores the fundamental concepts of inter-process communication within operating systems, with a specific focus on signals. We'll examine how signals facilitate communication between the operating system and running processes, as well as between different processes. Through practical examples and code demonstrations, we'll learn how to implement signal handlers, understand signal behavior, and manage signal masking for reliable program execution.

## Core Concepts Review

### Kernel Mode

When the CPU is in kernel mode, it is assumed to be executing trusted software, and thus it can execute any instructions and reference any memory address. The kernel is the core of the OS and it has complete control over everything that occurs in the system. The kernel is trusted software; all other programs are considered untrusted.

### Processes and Interrupts

A process is an executing instance of a program. An active process is a process that is currently advancing in the CPU while others are waiting in memory for their turns to use the CPU. The execution of a process can be interrupted by an interrupt. An interrupt is a notification to the OS that an event has occurred which results in changes in the sequence of instructions that is executed by the CPU.

### Types of Interrupts

- **Hardware/External interrupts** are those in which the notification originates from a hardware device.
- **Software/Internal interrupts** include exceptions and traps:

1. **Exceptions:** Similar to hardware interrupts but not caused by an external source; they occur during execution when an error occurs (e.g., division by 0, access to unmapped memory, etc.)
2. **Traps:** Occur in the usual run of the program, but unlike exceptions, they are not the product of some error. They are the execution of an instruction that is intended for user programs and transfers control to the OS. Such a request to the kernel is called a system call.

## Communication Mechanisms in Operating Systems

| **Mechanism** | **From** | **To** | **Purpose** |
|---|---|---|---|
| Interrupts | Hardware/Software | OS | Notify OS of events |
| System Calls | User Programs | OS | Request OS services |
| Signals | OS or Processes | Processes | Notify processes of events |

Table 1: Comparison of OS Communication Mechanisms

## Signals

Interrupts are communications between the CPU and the OS, and system calls are communications between users/programs and the OS. In contrast, signals are what the OS uses to communicate with processes; signals are also used for communication between different processes.

### What are Signals

Signals are notifications sent to/by processes or by the kernel to a process, providing a primitive communication mechanism between processes. Typically, signals are used to notify of the occurrence of events. Signals cause the process to stop whatever it is doing and force the process to handle them immediately.

### Difference between Signals and Interrupts

Interrupts are generated by the hardware or software and received and handled by the OS. Signals are generated by the OS or other processes, and received and handled by a process.

### Triggers for Signals

Signals can be triggered by various events in the system:

- Some examples for signal triggers:
  - Asynchronous input from the user such as `^C` (SIGINT), or typing ‘kill pid’ at the shell

  - The system or another process, for instance if an alarm set by the process has timed out (SIGALRM)
  - A software interrupt caused by an illegal instruction
    * The illegal instruction causes a software interrupt.
    * The software interrupt is received by the OS.
    * The OS generates a signal and sends it to the process.

By typing kill -l, we can see the whole list of signals from the command line.

Different signals have different ways of taking care of them, some to crash the program, some to stop it, etc.

## Sending a Signal From One Process to Another

- Signals can be used to send messages from one process to another.
- This is done by using

```
int kill(pid_t pid, int sig)
```

  - The `kill()` function sends a signal to a process specified by `pid`.
  - The `sig` parameter specifies which signal to send.
  - Despite its name, `kill()` doesn't necessarily terminate the process.
- The messages that are sent in this manner are predefined
  - We cannot send any data
  - Only the signal type itself carries meaning.
- SIGUSR1, SIGUSR2
  - These are user-defined signals that programs can use for their own purposes.
  - They allow applications to define custom responses to these signals.

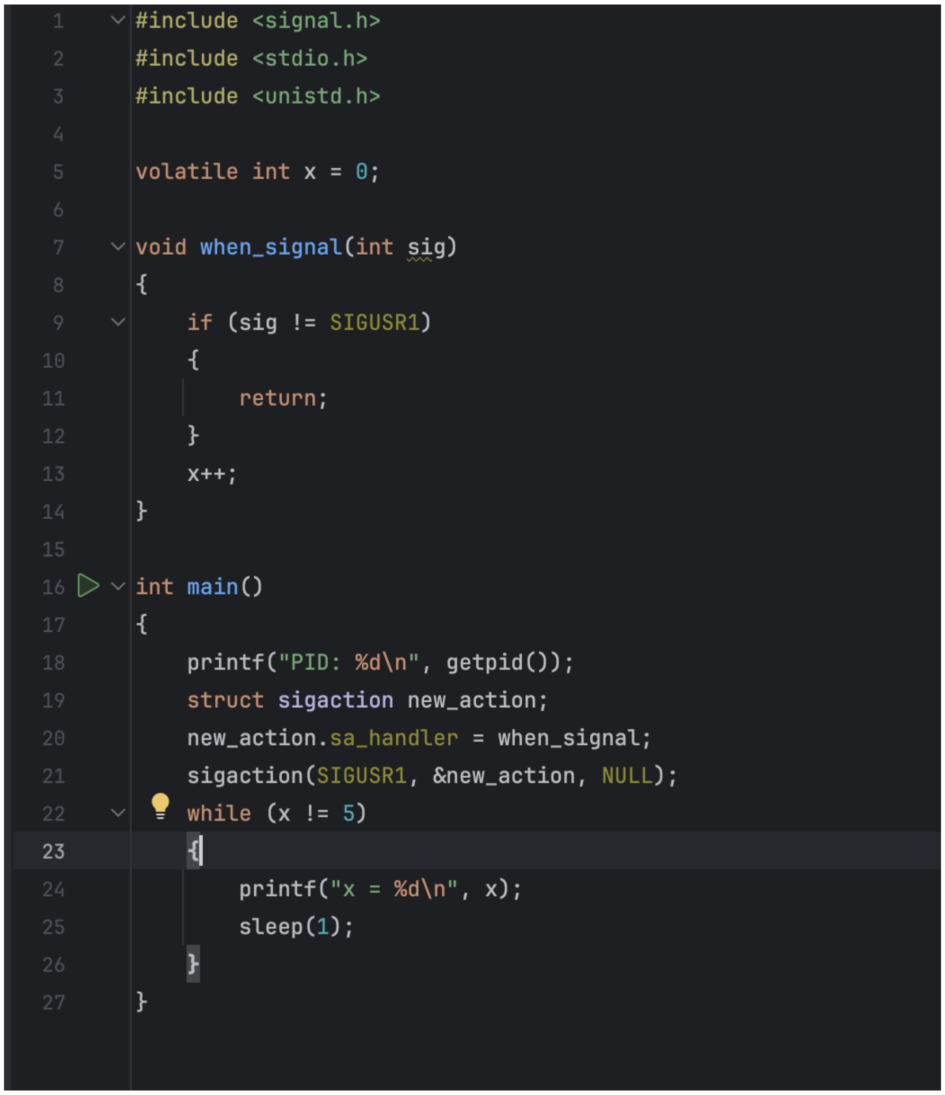
## Signal Handling

A process must run to handle given signals.

There are several types of handling for signals:

- The process can terminate, ignore, stop or continue executions.
- The process can specify which action to take when a signal is received, that is, executing a signal handler.
- There are, however, some signals that the process cannot catch, for example, KILL or STOP, and we cannot override them.
- If we do not install any signal handlers of our own, the runtime environment sets up a set of default signal handlers.

## Signal Handler Implementation: Example 1



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

volatile int x = 0;

void when_signal(int sig)
{
    if (sig != SIGUSR1)
    {
        return;
    }
    x++;
}

int main()
{
    printf("PID: %d\n", getpid());
    struct sigaction new_action;
    new_action.sa_handler = when_signal;
    sigaction(SIGUSR1, &new_action, NULL);
    while (x != 5)
    {
        printf("x = %d\n", x);
        sleep(1);
    }
}
```

Figure 1: Sigaction Example 1

**Overall Goal:** This program is designed to run, print its unique Process ID (`PID`), and then wait until it receives a specific signal (`SIGUSR1`) exactly five times from outside the program. Every second, it checks how many times it has received the signal and prints the current count before pausing again. It demonstrates basic signal handling in C.

**Global Variable:** `volatile int x = 0;`:

- `int x = 0;`: Declares an integer variable named `x` and initializes it to 0. This variable acts as a counter for the received signals.
- `volatile`: This is a crucial type qualifier. It tells the compiler that the value of `x` can change at any time in ways the compiler cannot predict (specifically, it will be modified asynchronously by the signal handler). This prevents the compiler from

making optimizations that might assume `x` only changes within the main loop's control flow, ensuring the program always reads the current, actual value of `x` from memory when checking the loop condition.

**Signal Handler Function (`when_signal`): `void when_signal(int sig)`:**

- Defines a function named `when_signal`. This function is designated to be executed automatically whenever the program receives a signal it's configured to handle.
- `int sig`: It takes one argument, `sig`, which holds the integer number identifying the signal that occurred.
- `if (sig != SIGUSR1)`: It checks if the signal received is specifically `SIGUSR1`. `SIGUSR1` is a "user-defined signal 1," intended for custom use by programmers.
- `return;`: If the signal is *not* `SIGUSR1`, the function does nothing and returns immediately.
- `x++;`: If the signal *is* `SIGUSR1`, this line increments the global counter variable `x` by one.

**The `main` Function:** This is the entry point where the program execution begins.

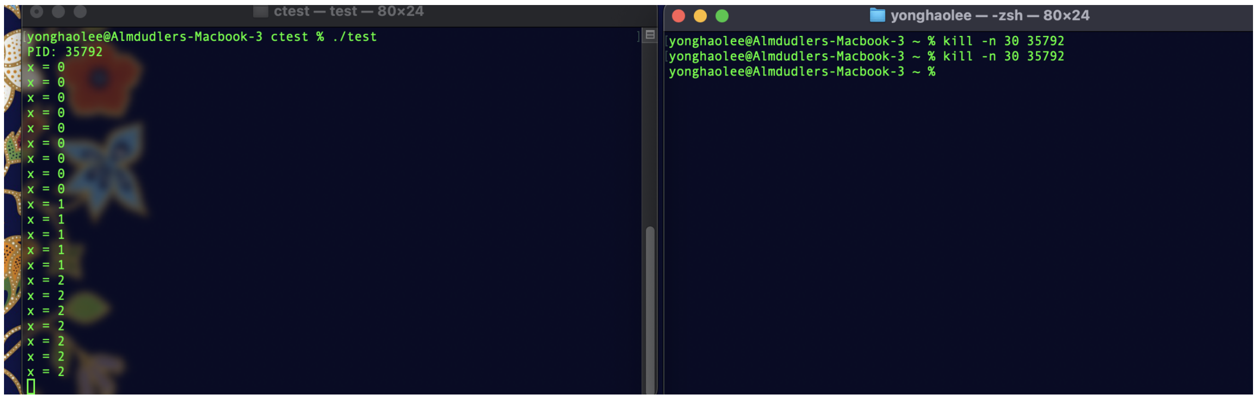
- `printf("PID: %d`
  `n", getpid());`:
  - `getpid()`: Calls the function to retrieve the unique Process ID (PID) assigned to this running instance of the program by the operating system.
- **Setting up the Signal Handler**:
  - `struct sigaction new_action;`: Declares a variable `new_action` of type `struct sigaction`. This structure is used to specify detailed signal handling behavior.
  - `new_action.sa_handler = when_signal;`: Sets the `sa_handler` field of the structure. This specifically assigns the `when_signal` function as the handler to be called when the associated signal arrives.
  - `sigaction(SIGUSR1, &new_action, NULL);`: This function call registers the signal handler with the operating system.
    - `SIGUSR1`: Specifies that the rule being set is for the `SIGUSR1` signal.
    - `&new_action`: Passes the address of the filled structure containing the handling instructions (i.e., use `when_signal`).
    - `NULL`: Indicates that we don't need to retrieve the previous action associated with `SIGUSR1`.

    After this call, the OS knows to execute `when_signal` whenever this process receives `SIGUSR1`.
- **The Waiting Loop**:

- `while (x != 5)`: Starts a loop that continues indefinitely as long as the value of the global variable `x` is not equal to 5.
- `printf("x = %d`
  `n", x);`: Inside the loop, this prints the current value of the counter `x` to the console.
- `sleep(1);`: Pauses the execution of the main program thread for approximately 1 second.
- When `x` eventually becomes 5 (because `when_signal` has been executed 5 times due to receiving 5 `SIGUSR1` signals), the loop condition `x != 5` becomes false, the loop terminates, the `main` function ends, and the program exits.

**How it Works in Practice:**

1. Compile the C code (e.g., `gcc your_code.c -o signal_counter`).
2. Run the compiled executable (`./signal_counter`).
3. The program will print its PID, for example: `PID: 12345`. Note this number.
4. It will then start printing `x = 0` once per second.
5. Open a *second* terminal window.
6. In the second terminal, use the `kill` command to send the `SIGUSR1` signal to the running program, using the PID you noted: `kill -SIGUSR1 12345` (replace `12345` with the actual PID).
7. Observe the first terminal. The signal handler increments `x`, and soon the output will change to `x = 1`.
8. Repeat step 6 four more times. Each time you send the signal, `x` will increase (`x = 2`, `x = 3`, `x = 4`).
9. After you send the fifth `SIGUSR1` signal, `x` becomes 5. The loop condition `x != 5` becomes false, and the program will terminate cleanly.

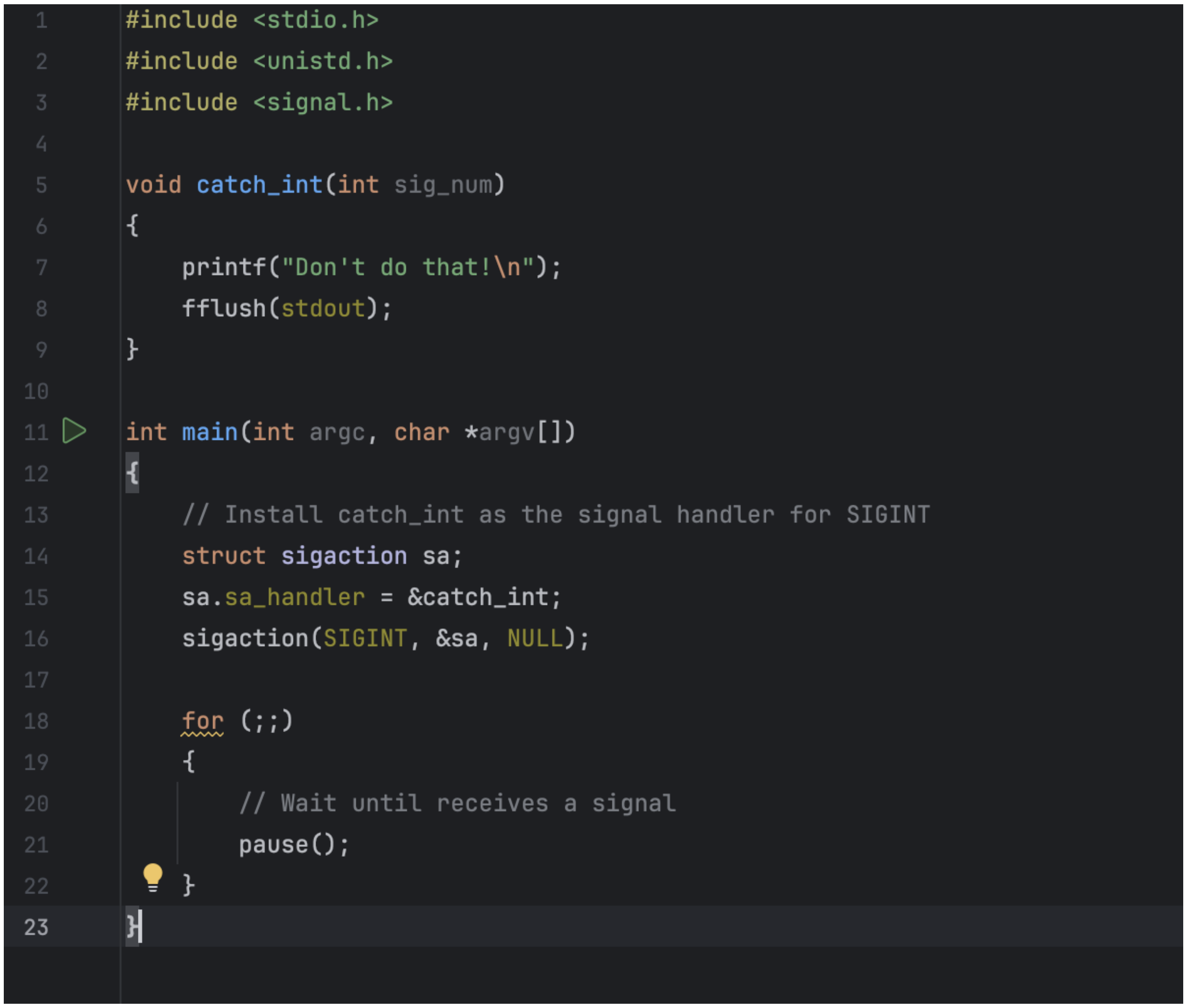


```
yonghaolee@Almdudlers-Macbook-3 ctest % ./test
PID: 35792
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 0
x = 1
x = 1
x = 1
x = 1
x = 1
x = 2
x = 2
x = 2
x = 2
x = 2
x = 2
x = 2
```

```
yonghaolee@Almdudlers-Macbook-3 ~ % kill -n 30 35792
yonghaolee@Almdudlers-Macbook-3 ~ % kill -n 30 35792
yonghaolee@Almdudlers-Macbook-3 ~ %
```

Figure 2: Sending signals from terminal

## Signal Handler Implementation: Example 2



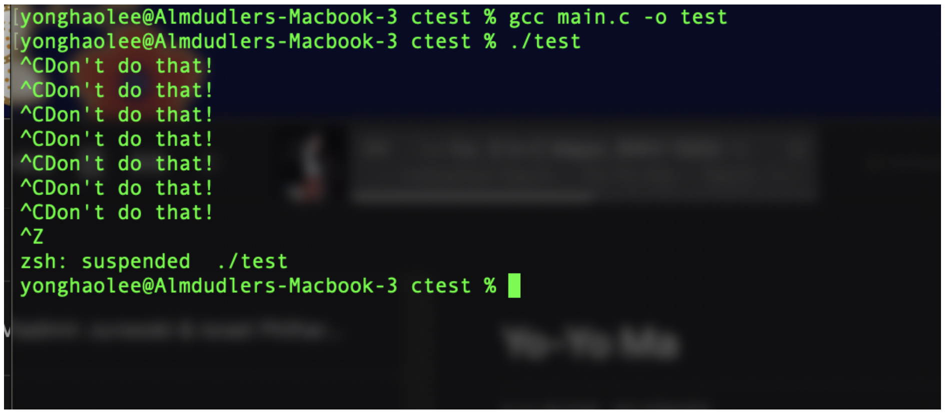
```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void catch_int(int sig_num)
{
    printf("Don't do that!\n");
    fflush(stdout);
}

int main(int argc, char *argv[])
{
    // Install catch_int as the signal handler for SIGINT
    struct sigaction sa;
    sa.sa_handler = &catch_int;
    sigaction(SIGINT, &sa, NULL);

    for (;;)
    {
        // Wait until receives a signal
        pause();
    }
}
```

Figure 3: Sigaction example 2

This program demonstrates overriding the default termination behavior for the interrupt signal (`SIGINT`), typically generated by Ctrl+C.

- **Functionality:** Catches `SIGINT` signals, prints a message, and continues waiting indefinitely.
- **Handler (`catch_int`):** Activated by `SIGINT`, prints "Don't do that!" and uses `fflush(stdout)` to guarantee the message appears immediately.
- **Main Loop:** Enters an infinite `for (;;)` loop containing only `pause()`. The `pause()` system call efficiently suspends the process until any signal (that isn't ignored) is caught.

- **Mechanism:** `sigaction` registers `catch_int` for `SIGINT`, replacing the default action (termination). Other signals (`SIGTSTP`/`Ctrl+Z`, etc.) retain their default behavior.



```
yonghaolee@Almdudlers-Macbook-3 ctest % gcc main.c -o test
yonghaolee@Almdudlers-Macbook-3 ctest % ./test
^CDon't do that!
^CDon't do that!
^CDon't do that!
^CDon't do that!
^CDon't do that!
^CDon't do that!
^CDon't do that!
^Z
zsh: suspended  ./test
yonghaolee@Almdudlers-Macbook-3 ctest %
```

Figure 4: Overriding SIGINT

### Predefined Signal Handlers: SIG_IGN and SIG_DFL

Instead of writing a custom function (like `catch_int` or `when_signal`) to handle a signal, the system provides two predefined constant values that can be used as signal handlers when calling functions like `signal()` or setting the `sa_handler` field in `sigaction`:

- `SIG_IGN` (Signal Ignore):
  - **Purpose:** Instructs the operating system to completely ignore and discard the specified signal when it arrives.
- `SIG_DFL` (Signal Default):
  - **Purpose:** Instructs the operating system to perform its standard, default action for the specified signal.

## Masking Signals

### Motivation

Assume that a process performs a cleanup. If during the cleanup the program exits abruptly, some old files will remain, resulting in data corruption. In order to avoid this situation, signals that can cause the process to exit like SIGINT should be blocked during cleanup.

### Signal Process Mask (sigprocmask)

`sigprocmask` allows a process to specify a set of signals to block and/or retrieve the list of signals that were previously blocked.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
```

**int how:** Determines the action to take:

- Add (`SIG_BLOCK`): Adds the signals in `set` to the current mask
- Delete (`SIG_UNBLOCK`): Removes the signals in `set` from the mask
- Set (`SIG_SETMASK`): Replaces the current mask with `set`

**const sigset_t *set:** The set of signals to be used in the operation

**sigset_t *oldset:** If not `NULL`, the previous mask will be returned here

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGTERM);
sigprocmask(SIG_SETMASK, &set, NULL);
//blocked signals: SIGINT and SIGTERM
sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGALRM);
sigprocmask(SIG_BLOCK, &set, NULL);
//blocked signals: SIGINT, SIGTERM, SIGALRM

sigemptyset(&set);
sigaddset(&set, SIGTERM);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_UNBLOCK, &set, NULL);
//blocked signals: SIGINT and SIGALRM
```

Figure 5: Signal Process Mask

**Example Code Analysis:**

**Example 1 - Set:** Uses `SIG_SETMASK` to replace the current mask

- Initializes an empty signal set
- Adds SIGINT and SIGTERM to the set
- Sets this as the new mask, completely replacing any previous masks
- Result: Only SIGINT and SIGTERM are blocked

**Example 2 - Add:** Uses `SIG_BLOCK` to add to the current mask

- Initializes an empty signal set
- Adds SIGINT and SIGALRM to the set
- Blocks these signals in addition to already blocked signals
- Result: SIGINT, SIGTERM (from before), and SIGALRM are now blocked

**Example 3 - Delete:** Uses `SIG_UNBLOCK` to remove from the mask

- Initializes an empty signal set
- Adds SIGTERM and SIGUSR1 to the set
- Unblocks these signals from the current mask
- Result: SIGINT and SIGALRM remain blocked, but SIGTERM is now unblocked

### Signal Blocking Rationale

- **Because signals are handled asynchronously**, race conditions can occur:
  - A signal may be received and handled in the middle of an operation that should not be interrupted
  - A second signal may occur before the current signal handler finished
    - The second signal may be of a different type or of the same type as the first one
- **Therefore we need to block signals** from being processed when they are harmful
  - The blocked signal will be processed after the block is removed
  - Some signals cannot be blocked (e.g., `SIGKILL`, `SIGSTOP`)

## Threads

### What is a Thread

A process is managed by the OS and is represented by a Process Control Block (PCB). The PCB contains:

1. Process data:
   - registers (PC, SP etc.)
   - memory (data, heap, stack and text)
   - environment (files etc.)
2. OS data:
   - priority-relevant data (time, priority, resources etc.)

- the user - access rights
- state

In the past, we only had one processor with one core, and it operated tasks sequentially. Today, we have more cores, which are components that share the processor's cache. We can think of cores as small processors within the processor, though there is an upper limit to how many cores we can put in a processor.

## Context Switching Overhead

The Figure below illustrates the overhead associated with context switching:

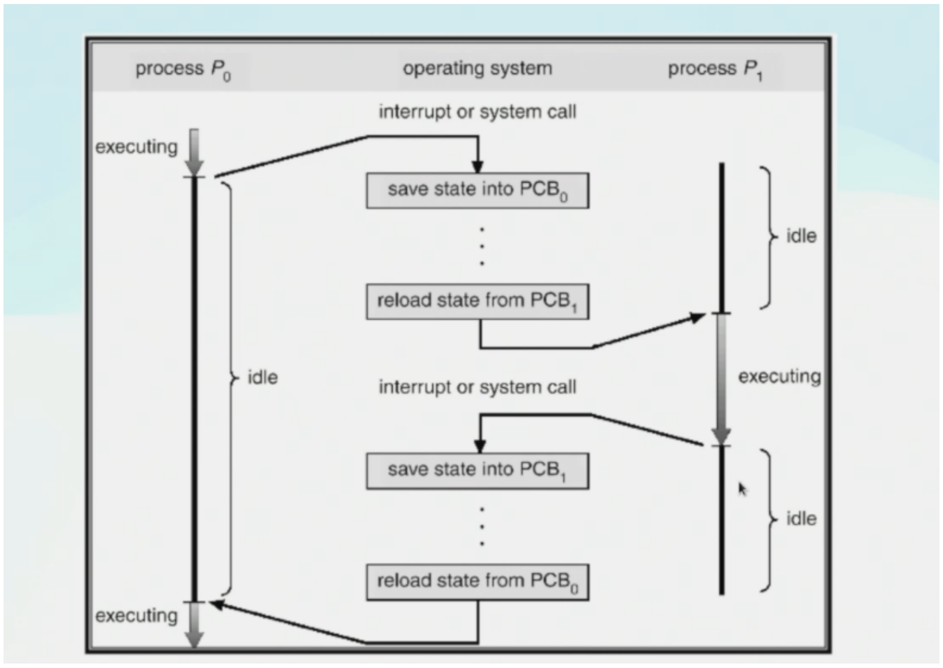


Figure 6: Context switch

- Between each process execution, there is a period where the CPU is performing administrative work rather than executing user processes
- This overhead consists of:
  - Time spent saving the state of the currently running process into its PCB
  - Time spent reloading the state of the next process from its PCB
  - The transition periods where neither process is executing (marked as "idle" in the diagram)
- These overhead operations consume CPU cycles that could otherwise be used for productive work
- The diagram shows that for each context switch, there are four distinct operations:

1. Save state from running process
2. Transition period (with dots in the diagram indicating other potential OS operations)
3. Reload state for the next process
4. Transfer control to the next process

## Thread Characteristics

A thread lives within a process, and a process can have several threads.

A thread possesses an independent flow of control, meaning it executes a sequence of instructions independently. It can be scheduled to run separately from other threads within the same process since it maintains its own:

- Stack (for function calls, local variables, and return addresses)
- Registers (CPU state including program counter and stack pointer)

The other resources of the process are shared by all its threads, including:

- Code section (program instructions)
- Data section (global variables)
- Heap memory (dynamically allocated memory)
- Open files

This sharing of resources makes thread creation more efficient than process creation and facilitates communication between threads, as they can directly access the same memory space without needing special inter-process communication mechanisms.

## Thread Implementations

There are two types of threads: user-level and kernel-level threads (lightweight processes). The distinction is based on who manages them. User-level threads are managed by the user, whereas kernel-level threads are managed by the kernel. Note that kernel-level threads are not the same as the threads of the kernel itself.

## User-Level Threads

User-level threads are implemented as a thread library that executes entirely in user space. This library contains the necessary code for:

- Thread creation
- Thread termination
- Thread scheduling

- Context switching between threads

In this implementation, the operating system views the application as a single process, remaining unaware of the multiple threads operating within it. All thread management operations occur within the application's address space without kernel intervention. The thread library provides the API that applications use to create and manage their threads, with all thread operations being handled by library functions rather than system calls.

Switching from thread to thread is done entirely in user space, resulting in a lower context switching penalty compared to kernel threads.

The scheduling depends on the specific implementation. A significant limitation is that if one thread is blocked by the kernel, the entire process becomes blocked.

When CPU access time is fairly divided between processes, the threads within a process must share a single time quantum. This means all threads in the process collectively receive the same amount of CPU time as a single thread would in a different process.

Another consideration is fault isolation: if one thread crashes, the entire process crashes, as the threads share the same address space and lack the protection boundaries present in kernel-level implementations.

In user-level threads, only one thread can modify a shared resource at a time (if implemented correctly), so some of the locks required for threads may not be needed. Still, be careful!

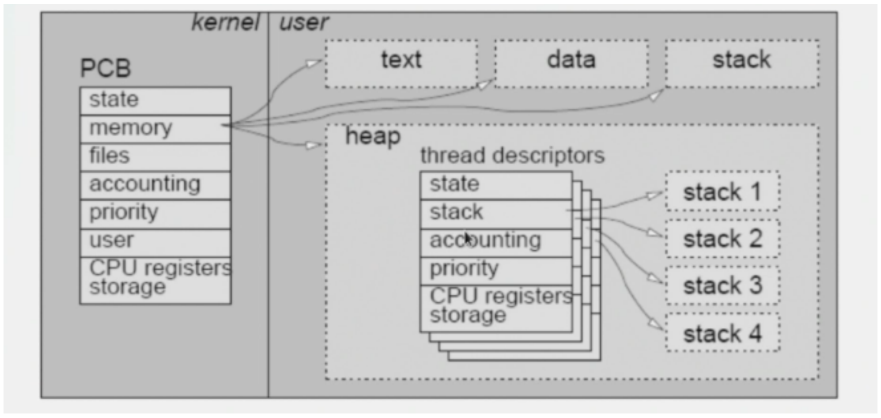


Figure 7: Thread descriptor

For each thread, the library maintains a thread descriptor in the application's heap, which functions similarly to a PCB but at the user level. As shown in the diagram, while the kernel only sees one PCB for the entire process (containing state, memory, files, accounting, priority, user, and CPU registers storage), the user space contains multiple thread descriptors. Each thread descriptor stores its own state, stack pointer, accounting information, priority, and CPU register values, with each thread having its own dedicated stack (stack 1, 2, 3, 4). This structure allows the thread library to manage threads independently while appearing as a single process to the kernel.

## Implementing a Thread Library

The process of switching between threads requires a specific sequence of operations:

1. **Stop** running the current thread
2. **Save** the current state of the thread
3. **Jump** to another thread and continue from where it stopped before, using its saved state

This context-switching mechanism relies on two specialized functions:

- `sigsetjmp` – Saves the current execution context, including:
  - Current location in the program
  - CPU register state
  - Signal mask
- `siglongjmp` – Performs a non-local jump to the saved location:
  - Restores the previously saved execution context
  - Resets CPU state to what it was at save time
  - Restores the signal mask

These functions provide the foundation for implementing cooperative multitasking, allowing a thread to voluntarily yield control to other threads in a controlled manner.

### sigsetjmp – save a "bookmark"

`sigsetjmp(sigjmp_buf env, int savesigs)`

- Saves stack context and CPU state in `env` for later use
- If `savesigs` is non-zero, saves current signal mask (in `env`)
  - When `savesigs` is non-zero, blocked signals remain blocked upon context restoration
- Later jump to this location using `siglongjmp`
- Return value:
  - 0 if returning directly from `sigsetjmp`
  - User-defined value if arrived via `siglongjmp`

### siglongjmp — use a “bookmark”

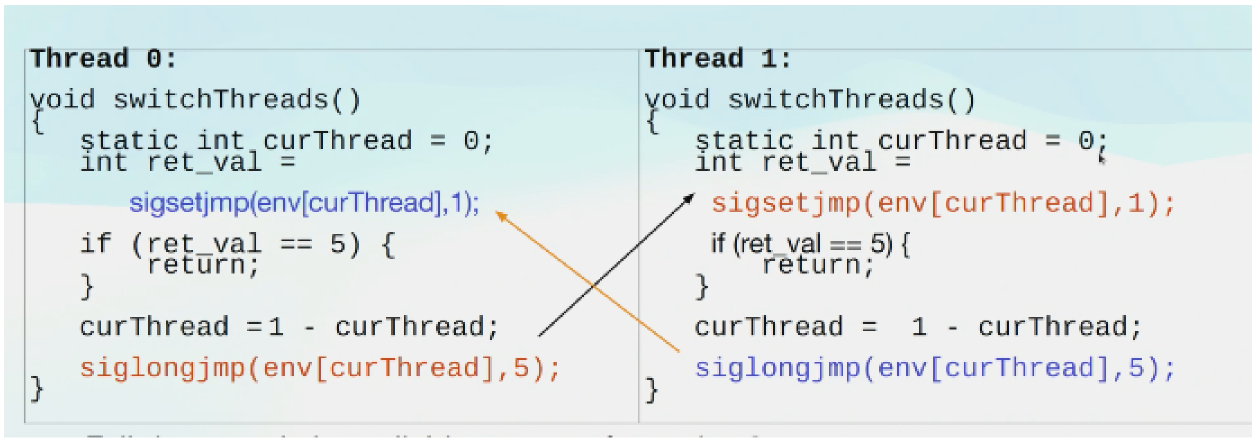
`siglongjmp(sigjmp_buf env, int val)`

- Jumps to the code location and restores CPU state specified by `env`
- The jump will take us into the location in the code where the `sigsetjmp` has been called
- If the signal mask was saved in `sigsetjmp`, it will be restored as well
- The return value of `sigsetjmp` after arriving from `siglongjmp`, will be the user-defined `val`

### What’s Saved in env

| **Saved** | **Not Saved** |
|---|---|
| Program counter (PC) | Global variables |
| Stack pointer (SP) | Dynamically allocated variables |
| Local variable locations | Actual values of local variables |
| Return addresses | Global resources |
| Signal mask (if specified) | |
| CPU registers & state | |

### The Switch in Exercise 2



```
Thread 0:
void switchThreads()
{
    static int curThread = 0;
    int ret_val =
        sigsetjmp(env[curThread],1);
    if (ret_val == 5) {
        return;
    }
    curThread =1 - curThread;
    siglongjmp(env[curThread],5);
}
```

```
Thread 1:
void switchThreads()
{
    static int curThread = 0;
    int ret_val =
     sigsetjmp(env[curThread],1);
     if (ret_val == 5) {
        return;
    }
    curThread =  1 - curThread;
    siglongjmp(env[curThread],5);
}
```

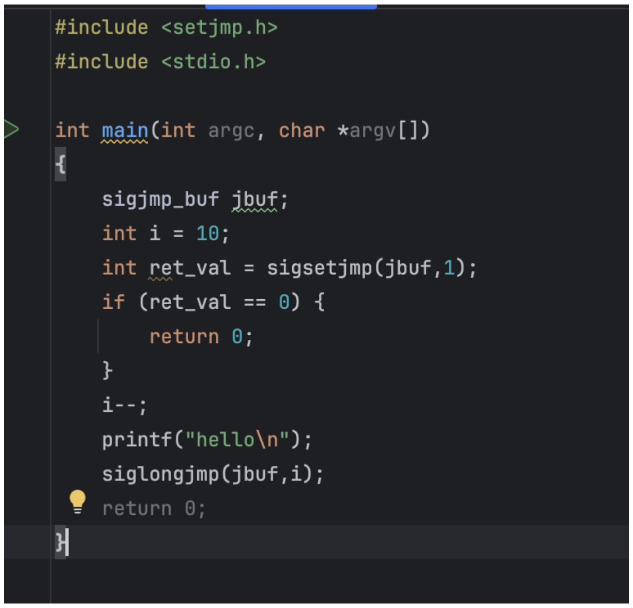
Figure 8: The switch

- Both threads use identical `switchThreads()` functions that cooperatively pass control
- `env` is an array of `sigjmp_buf` structures that store saved execution contexts
- `env[0]` stores Thread 0’s state, `env[1]` stores Thread 1’s state

- `static int curThread` tracks which thread is currently active (0 or 1)
- `sigsetjmp(env[curThread], 1)` saves the current thread's state
- `curThread = 1 - curThread` toggles to the other thread's ID
- `siglongjmp(env[curThread], 5)` jumps to the other thread's saved context
- The `if (ret_val == 5)` check prevents endless back-and-forth switching

### Exam Question - Context Switch

How many times is *hello* printed?



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

int main(int argc, char *argv[])
{
    sigjmp_buf jbuf;
    int i = 10;
    int ret_val = sigsetjmp(jbuf,1);
    if (ret_val == 0) {
        return 0;
    }
    i--;
    printf("hello\n");
    siglongjmp(jbuf,i);
    return 0;
}
```

Figure 9: Exam question

1. First, a `sigjmp_buf` variable `jbuf` is declared to store the execution context.
2. The variable `i` is initialized to 10.
3. The program calls `sigsetjmp(jbuf, 1)` which does two things:
    - Saves the current execution context in `jbuf`
    - Returns 0 for this initial invocation
4. The return value of `sigsetjmp` (which is 0) is stored in `ret_val`.
5. The program then checks if `ret_val == 0`, which is true in this first (and only) invocation.
6. Since the condition is true, it executes `return 0;` which immediately terminates the program.

7. The statements `i--`, `printf("hello")`, and `siglongjmp(jbuf,i)` are never reached.

`sigsetjmp` has the following behavior:

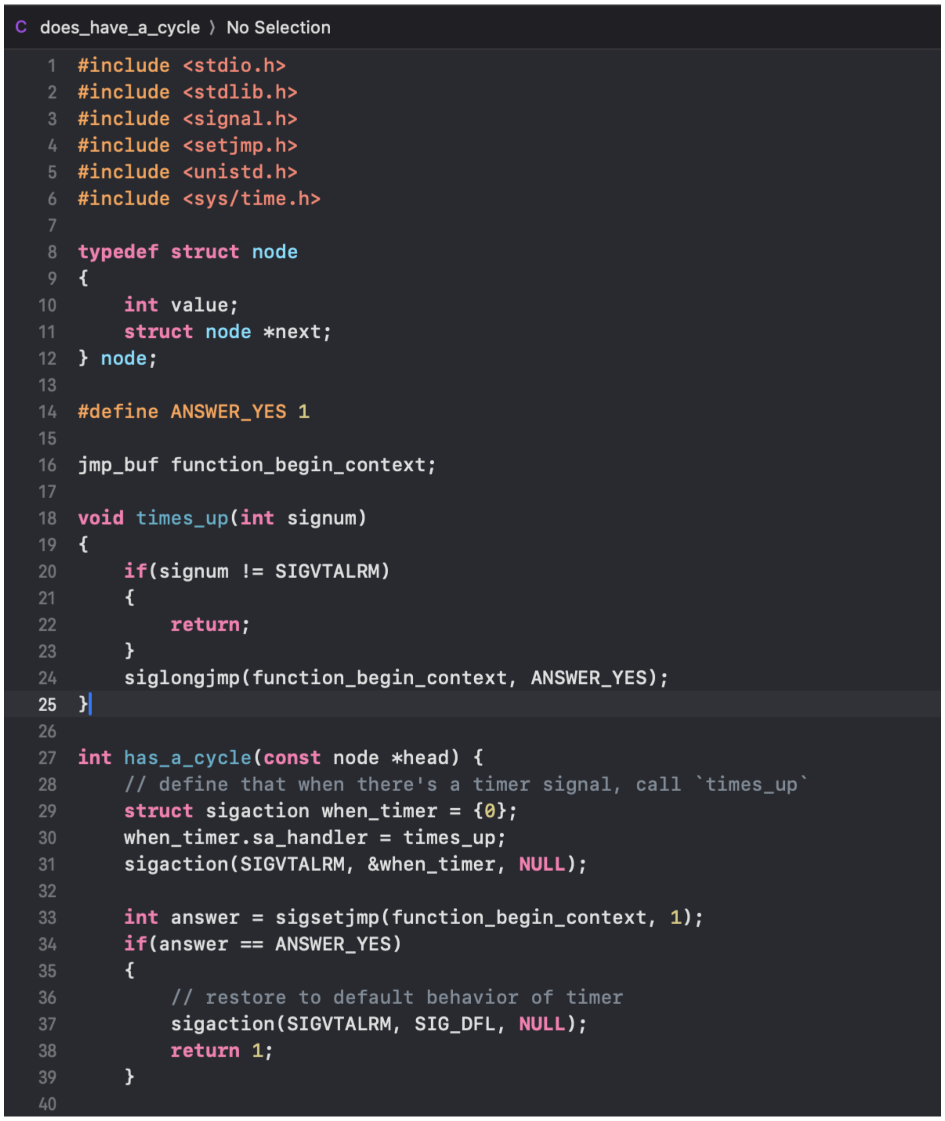
- When called directly: Returns 0
- When program control is returned to it via `siglongjmp(jbuf, val)`: Returns the value specified by the `val` parameter of `siglongjmp`

In this program, the condition `if (ret_val == 0)` causes an early exit before any printing can occur.

## Advanced Example - Cycle Detection Using Signals and Timers

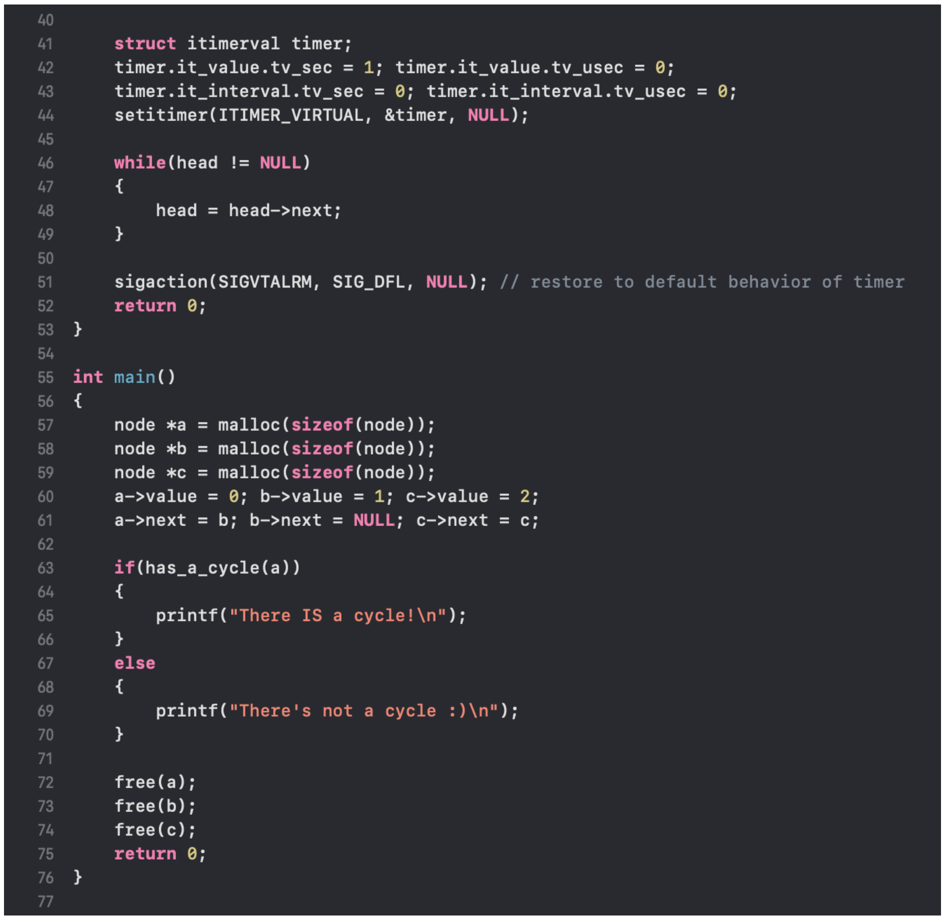
The following example uses signals and timers to detect whether a linked list has a cycle.

This code detects cycles in a linked list using a clever approach: if the list has a cycle, traversing it would run forever. Instead of using the traditional slow/fast pointer approach, this code uses a timer signal to interrupt the traversal if it takes too long, which indicates a cycle.



```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>
#include <unistd.h>
#include <sys/time.h>

typedef struct node
{
    int value;
    struct node *next;
} node;

#define ANSWER_YES 1

jmp_buf function_begin_context;

void times_up(int signum)
{
    if(signum != SIGVTALRM)
    {
        return;
    }
    siglongjmp(function_begin_context, ANSWER_YES);
}

int has_a_cycle(const node *head) {
    // define that when there's a timer signal, call `times_up`
    struct sigaction when_timer = {0};
    when_timer.sa_handler = times_up;
    sigaction(SIGVTALRM, &when_timer, NULL);

    int answer = sigsetjmp(function_begin_context, 1);
    if(answer == ANSWER_YES)
    {
        // restore to default behavior of timer
        sigaction(SIGVTALRM, SIG_DFL, NULL);
        return 1;
    }

```

Figure 10: Cycle Detector Using Timer 1



```c

    struct itimerval timer;
    timer.it_value.tv_sec = 1; timer.it_value.tv_usec = 0;
    timer.it_interval.tv_sec = 0; timer.it_interval.tv_usec = 0;
    setitimer(ITIMER_VIRTUAL, &timer, NULL);

    while(head != NULL)
    {
        head = head->next;
    }

    sigaction(SIGVTALRM, SIG_DFL, NULL); // restore to default behavior of timer
    return 0;
}

int main()
{
    node *a = malloc(sizeof(node));
    node *b = malloc(sizeof(node));
    node *c = malloc(sizeof(node));
    a->value = 0; b->value = 1; c->value = 2;
    a->next = b; b->next = NULL; c->next = c;

    if(has_a_cycle(a))
    {
        printf("There IS a cycle!\n");
    }
    else
    {
        printf("There's not a cycle :)\n");
    }

    free(a);
    free(b);
    free(c);
    return 0;
}

```

Figure 11: Cycle Detector Using Timer 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>
#include <unistd.h>
#include <sys/time.h>

typedef struct node {
    int value;
    struct node *next;
} node;

#define ANSWER_YES 1

jmp_buf function_begin_context;
```

**Explanation:**

- A simple linked list node structure is defined with an integer value and a pointer to the next node.

- `ANSWER_YES` is defined as a constant with value 1, used to indicate a positive result.
- `jmp_buf function_begin_context` is a global variable that will store the execution context for non-local jumps.

**Signal Handler Function**

```
void times_up(int signum) {
    if(signum != SIGVTALRM) {
        return;
    }
    siglongjmp(function_begin_context, ANSWER_YES);
}
```

**Explanation:**

- This function is a signal handler that gets called when a timer expires.
- It first verifies that the signal received is `SIGVTALRM` (virtual timer alarm). If it's any other signal, it simply returns and does nothing.
- If the signal is `SIGVTALRM`, it uses `siglongjmp` to jump back to a previously saved program point.
- `ANSWER_YES` (value 1) is passed as the return value for the corresponding `sigsetjmp` call.
- This non-local jump effectively interrupts the normal program flow when the timer expires.

**Cycle Detection Function**

```
int has_a_cycle(const node *head) {
    // define that when there's a timer signal, call ‘times_up‘
    struct sigaction when_timer = {0};
    when_timer.sa_handler = times_up;
    sigaction(SIGVTALRM, &when_timer, NULL);

    int answer = sigsetjmp(function_begin_context, 1);
    if(answer == ANSWER_YES) {
        // restore to default behavior of timer
        sigaction(SIGVTALRM, SIG_DFL, NULL);
        return 1;
    }
    struct itimerval timer;
    timer.it_value.tv_sec = 1; timer.it_value.tv_usec = 0;
    timer.it_interval.tv_sec = 0; timer.it_interval.tv_usec =
       0;
    setitimer(ITIMER_VIRTUAL, &timer, NULL);

    while(head != NULL) {
        head = head->next;
    }

    sigaction(SIGVTALRM, SIG_DFL, NULL); // restore to default
       behavior of timer
    return 0;
}
```

The function is broken down into sections:

**Signal Handler Registration**

```
struct sigaction when_timer = {0};
when_timer.sa_handler = times_up;
sigaction(SIGVTALRM, &when_timer, NULL);
```

- Creates a new `sigaction` structure initialized to zeros.
- Sets the signal handler field to our `times_up` function.
- Registers this handler with the operating system for the `SIGVTALRM` signal.
- This tells the OS: "When a virtual timer alarm occurs, call the `times_up` function."
- The `NULL` parameter indicates we don't need to store the previous handler.

**Context Saving and Jump Target**

```
int answer = sigsetjmp(function_begin_context, 1);
if(answer == ANSWER_YES) {
    // restore to default behavior of timer
    sigaction(SIGVTALRM, SIG_DFL, NULL);
    return 1;
}
```

- `sigsetjmp` saves the current execution state (registers, stack pointer, program counter) in `function_begin_context`.
- The first time this runs, it returns 0, so `answer` is 0 and the condition is false.
- If we later jump back here via `siglongjmp`, `answer` will be set to `ANSWER_YES` (1).

- The condition checks if we've arrived at this point via the signal handler.
- If true, we reset the signal handler to default behavior using `SIG_DFL` and return 1 (indicating a cycle was found).
- The parameter 1 in `sigsetjmp` means also save the signal mask.

**Timer Setup**

```
struct itimerval timer;
timer.it_value.tv_sec = 1; timer.it_value.tv_usec = 0;
timer.it_interval.tv_sec = 0; timer.it_interval.tv_usec = 0;
setitimer(ITIMER_VIRTUAL, &timer, NULL);
```

- Creates a timer structure of type `itimerval`.
- `it_value` specifies when the timer will first expire (1 second of CPU time, 0 microseconds).
- `it_interval` set to 0 means the timer will not repeat.
- `setitimer` activates the timer with type `ITIMER_VIRTUAL`, which counts only CPU time used by the process.
- When the timer expires after 1 second of CPU time, it will send a `SIGVTALRM` signal.
- The `NULL` parameter means we don't need information about any previous timer.

**List Traversal and Return**

```
while(head != NULL) {
    head = head->next;
}

sigaction(SIGVTALRM, SIG_DFL, NULL); // restore to default
   behavior of timer
return 0;
```

- A simple loop that traverses the linked list by following each `next` pointer.
- If the list has a cycle, this loop would run forever because `head` would never become `NULL`.
- If the timer expires during this loop, our signal handler will be called, which will jump back to the `sigsetjmp` call.
- If the loop completes normally (reaches the end of the list), we reset the signal handler to its default behavior.
- Returning 0 indicates "no cycle found" since we successfully reached the end of the list before the timer expired.

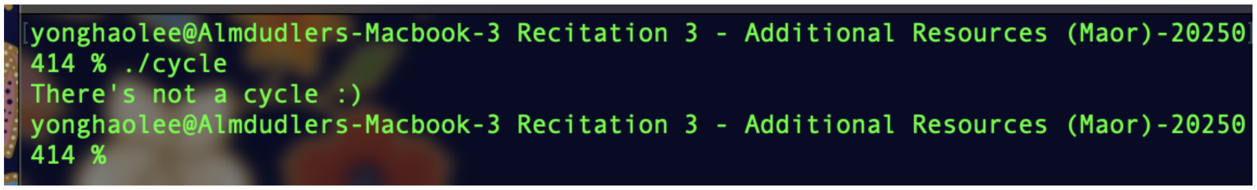
**Main Function**

```
int main() {
    node *a = malloc(sizeof(node));
    node *b = malloc(sizeof(node));
    node *c = malloc(sizeof(node));
    a->value = 0; b->value = 1; c->value = 2;
    a->next = b; b->next = NULL; c->next = c;

    if(has_a_cycle(a)) {
        printf("There IS a cycle!\n");
    }
    else {
        printf("There's not a cycle :)\n");
    }

    free(a);
    free(b);
    free(c);
    return 0;
}
```

**Explanation:**

- Creates three nodes: `a`, `b`, and `c`.
- Sets up two test cases:
  - A list `a->b->NULL` (no cycle)
  - A node `c` that points to itself (has a cycle)
- Calls `has_a_cycle` with the first list (`a`) which has no cycle.
- Prints the result based on the return value.
- Frees the allocated memory.

The code yields the following:



```
[yonghaolee@Almdudlers-Macbook-3 Recitation 3 - Additional Resources (Maor)-20250]
414 % ./cycle
There's not a cycle :)
yonghaolee@Almdudlers-Macbook-3 Recitation 3 - Additional Resources (Maor)-20250
414 %
```

Figure 12: Detection result

## Kernel-Level Threads

Kernel-level threads are represented by the kernel's own thread table and scheduled by the OS scheduler. If one thread blocks on I/O or a syscall, the kernel simply deschedules it and continues running other threads in the same process.

Switching between kernel-level threads is relatively expensive because it involves:

- A trap into the kernel and back to user space,
- Saving/restoring the full CPU context

*Note:* "Kernel threads" (sometimes called daemons) are threads that run kernel code in kernel mode; they're distinct from the user-created kernel-level threads described above.

## Comparing Thread Implementations

### User-Level Threads (Advantages)

- Very cheap context-switches (user-space only)
- Application-specific scheduling
- No kernel overhead for thread operations

### Kernel-Level Threads (Advantages)

- True parallelism across multiple cores
- Independent blocking per thread
- OS-managed fairness and priorities

### User-Level Threads (Disadvantages)

- Cannot utilize multiple cores
- One blocked thread blocks the entire process
- Performance tied to OS scheduler quality

### Kernel-Level Threads (Disadvantages)

- Higher context-switch overhead
- Must explicitly synchronize shared data

When choosing an implementation, one must consider the specifications and needs of the application.

For an application that switches between threads often, user-level threads may be better. For one that has many threads, or many that are I/O bound, kernel-level threads may be better. Often, a hybrid approach using both can be optimal.

## Exam Question

What is the difference between processes and kernel-level threads?

1. With multiple processes you can utilize many CPUs, but kernel-level threads cannot.
   **Incorrect.** Kernel-level threads are scheduled by the OS just like processes, and each can run on its own core.

2. A kernel-level thread prevents the whole program from blocking on I/O, but with multiple processes the blocking problem still exists.
   **Incorrect.**

   - First half correct: if one kernel-level thread blocks, the OS deschedules only that thread and lets others run.
   - Second half incorrect: each process is also an independent entity—blocking in one process does not block other processes.

3. Processes need OS mediation to communicate; kernel-level threads do not.
   **Correct.** Separate processes have isolated address spaces requiring IPC via the kernel, whereas threads share memory and communicate directly.

4. Kernel threads run only kernel code, whereas processes can run user code.
   **Incorrect.**

   - "Processes can run user code" is true: user-land processes execute in user mode.
   - "Kernel threads run only kernel code" is misleading here: user-created kernel-level threads also run user-mode code. The term "kernel threads" properly refers to OS daemons running in kernel mode.