Chapter 26: Concurrency - An Introduction

Overview

- **Main Idea**: Threads enable multiple points of execution within a single process, sharing the same address space but having private registers/stacks.
- **Key Terms**: Thread, Process, Address Space, Context Switch, Race Condition, Critical Section, Mutual Exclusion.

---

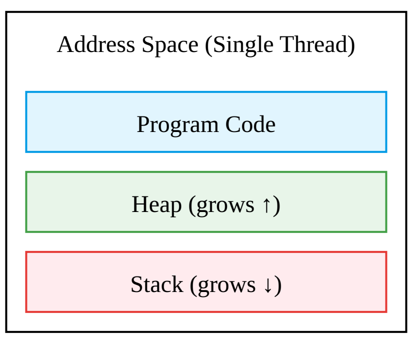
1. Threads vs Processes

Key Differences

Feature	Thread	Process
Address Space	Shared with other threads	Private
Context Switch	Faster (no page table switch)	Slower
Communication	Direct (shared memory)	Complex (pipes, messages)

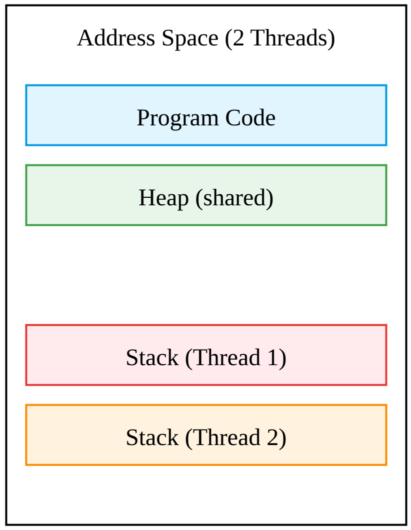
Address Space Layout

Single-Threaded Process



(Stack grows downward, heap upward.)

Multi-Threaded Process (2 threads)



*(Each thread has its own stack; heaps/code are shared.)_

---

2. Why Use Threads?

Parallelism Example

```
// Parallel array addition (pseudo-code)
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];  // Threads split work
}
```

I/O Overlap

```
// Web server handling concurrent requests
while (1) {
    int conn_fd = accept();  // Blocking call
    pthread_create(&thread, NULL, handle_request, (void*)conn_fd);
}
```

---

3. Thread Creation Code

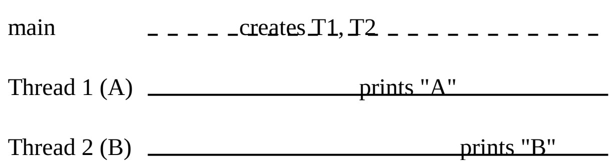
```
#include <pthread.h>
void *mythread(void *arg) {
```

```c
    printf("%s\n", (char *)arg);  // Thread-local work
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, mythread, "A");  // Create Thread 1
    pthread_create(&t2, NULL, mythread, "B");  // Create Thread 2
    pthread_join(t1, NULL);  // Wait for threads
    pthread_join(t2, NULL);
    printf("main: end\n");
}
```

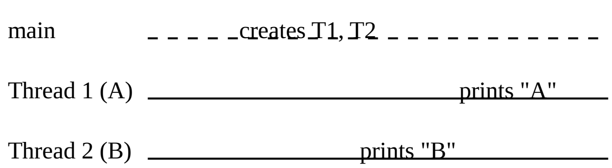
Possible Execution Traces

1. **Order 1**:



   (`main → A → B`)

2. **Order 2**:



   (`main → B → A`)

4. Race Conditions

Problematic Code

```c
static volatile int counter = 0;
void *mythread(void *arg) {
    for (int i = 0; i < 1e7; i++) {
        counter++;  // Race condition!
    }
}
```

- **Expected**: `20,000,000` (2 threads × 10M increments).
- **Actual**: Random (e.g., `12,345,678`) due to interleaving.

Assembly Breakdown

```
mov 0x8049alc, %eax  ; Load counter
add $0x1, %eax        ; Increment (race here!)
mov %eax, 0x8049alc   ; Store back
```

Thread 1	mov counter, %eax	add $1, %eax	mov %eax, counter
Thread 2	mov counter, %eax	add $1, %eax	mov %eax, counter

Final counter = 51 (expected 52)

*(Interrupts corrupt final value.)_

---

5. Critical Sections & Solutions

Atomic Operation Wish

```
memory-add 0x8049alc, $0x1   ; Hypothetical atomic increment
```

Synchronization Primitives

- **Mutexes**: `pthread_mutex_lock(&lock)`
- **Semaphores**: `sem_wait(&sem)`

---

6. Key Concepts

Term	Definition
Critical Section	Code accessing shared data (e.g., `counter++`).
Race Condition	Undefined behavior due to unsynchronized access.
Mutual Exclusion	Ensuring only one thread executes critical section at a time.

---

7. Why OS Cares

- **Kernel Data Structures**: Threads manage concurrent access to OS resources (e.g., file systems).
- **Synchronization**: Built via hardware (atomic ops) + OS (scheduler).

Homework Notes

1. `loop.s`:

```
./x86.py -t 1 -p loop.s -i 100 -R dx   # Single-threaded
./x86.py -t 2 -p loop.s -i 3 -r        # Race with randomness
```

2. **Critical Section Identification**:

Critical Section

mov counter, %eax

add $1, %eax

mov %eax, counter

*(Interrupts must avoid `counter++` region.)_

References

- Dijkstra, E. W. (1965). *Solution of a problem in concurrent programming control*.
- Gray, J. (1992). *Transaction Processing: Concepts and Techniques*.