Course 3: Shared Memory

Kristaps Dzonsons

08 December, 2011

Course site: *http://kristaps.bsd.lv/minicourse_12_2011*

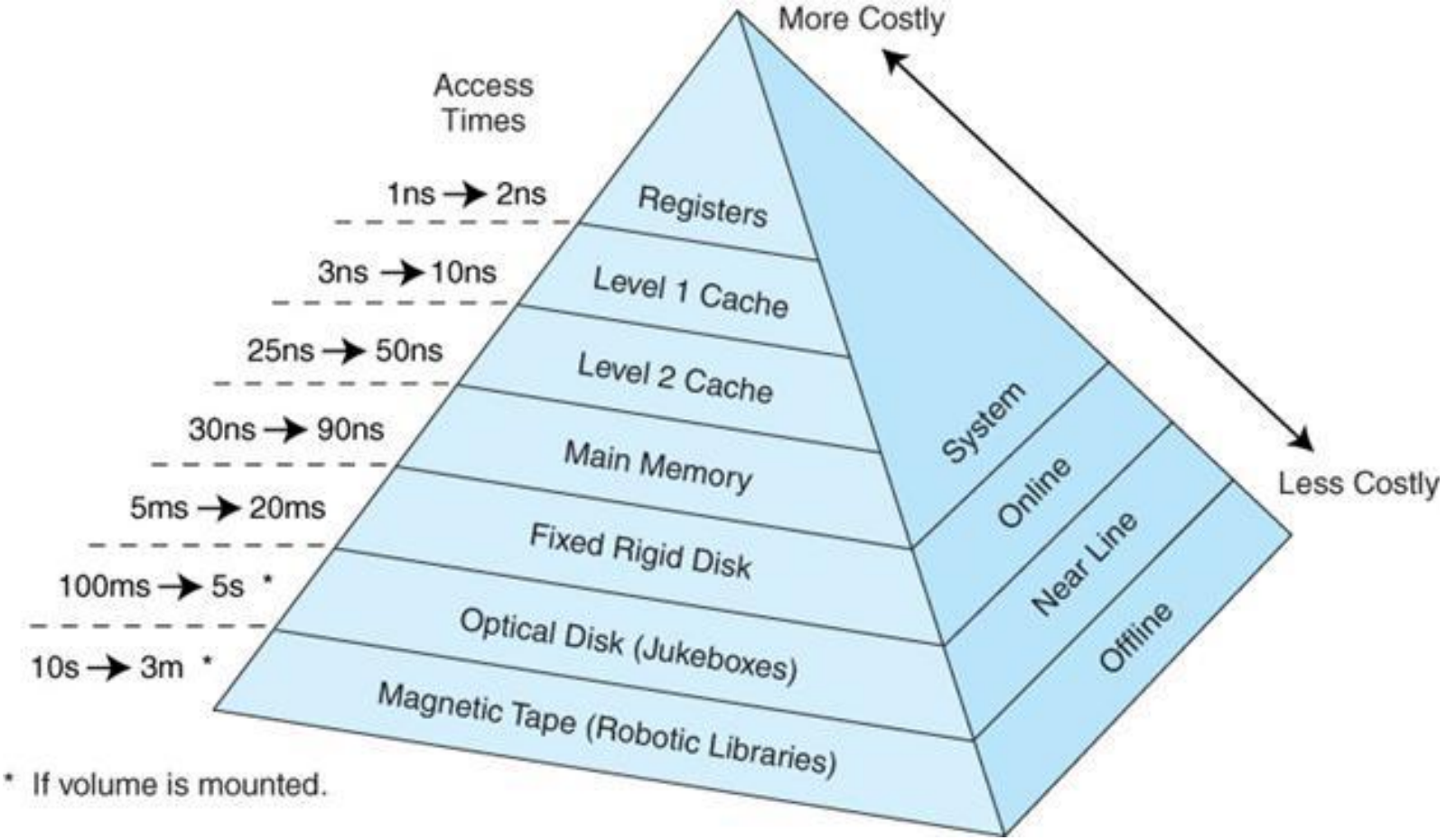
Environment

In prior lectures, we focussed on the environment of high-performance computing:

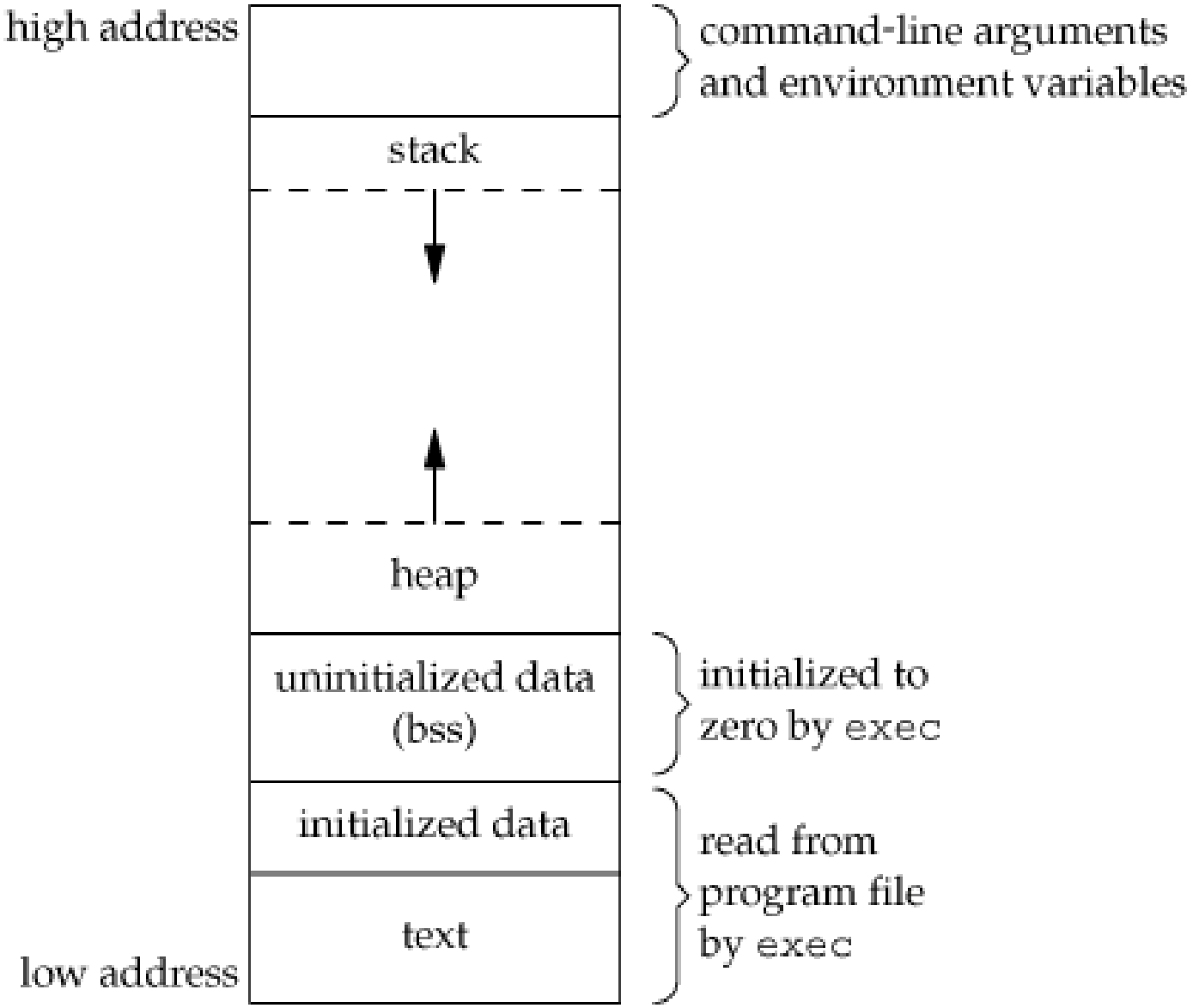
- **UNIX** an operating system popular in non-desktop environments
- **C** a low-level, minimal programming language
- **hardware** purpose-built, high-performance hardware

In this lecture, we consider C programming for high-performance hardware.

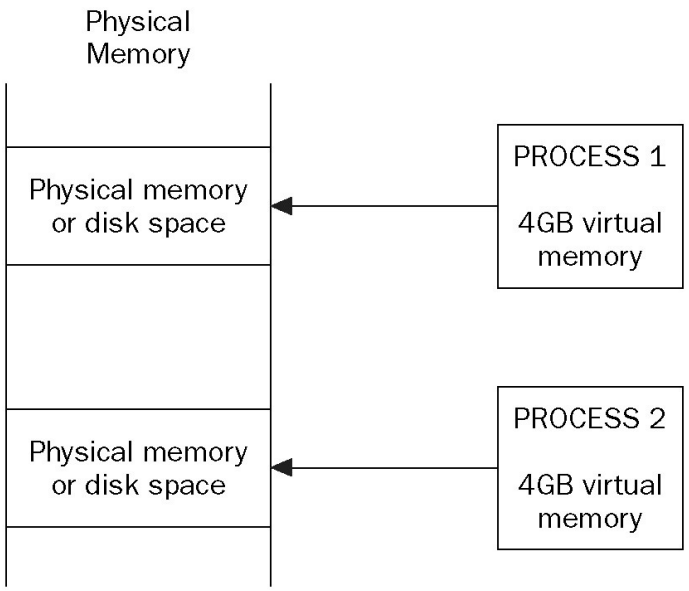
Review: Hardware

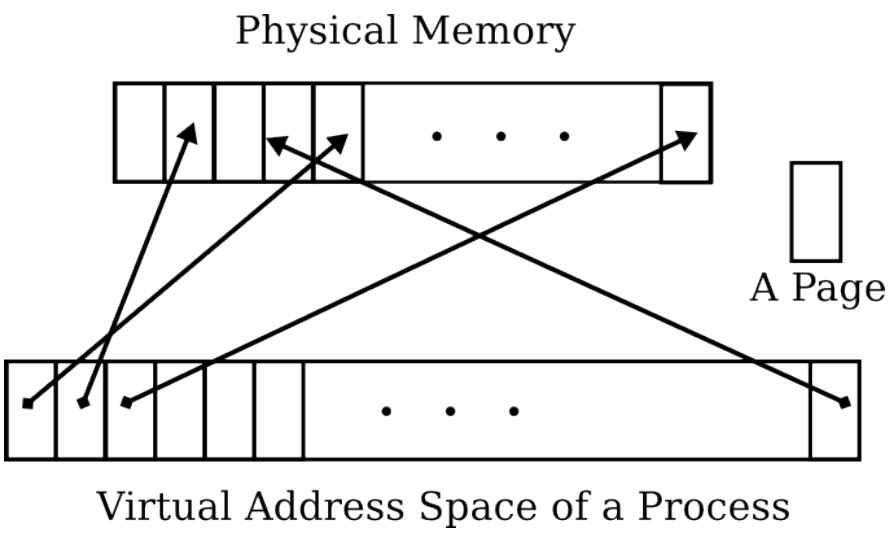


Review: Process Memory



Review: Process Memory





Review: Cache Missing

```
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
        int                 **matrix;
        int                   i, j;
        long long int         res;

        matrix = malloc(11000 * sizeof(int *));
        for (i = 0; i < 11000; i++)
                matrix[i] = malloc(11000 * sizeof(int));

        res = 0;
        for (i = 0; i < 11000; i++)
                for (j = 0; j < 11000; j++)
                        res += matrix[j][i];

        printf("%lld\n", res);
        return(EXIT_SUCCESS);
}
```

Review: Cache Missing

```
D1  misses:        7,584,406  (  7,571,336 rd   +  13,070 wr)
LLd misses:        7,584,016  (  7,570,986 rd   +  13,030 wr)
D1  miss rate:           0.7% (        0.7%     +     2.9%  )
LLd miss rate:           0.7% (        0.7%     +     2.9%  )
LL refs:           7,585,829  (  7,572,759 rd   +  13,070 wr)
LL misses:         7,585,370  (  7,572,340 rd   +  13,030 wr)
LL miss rate:            0.2% (        0.2%     +     2.9%  )

D1  misses:      136,140,031  (136,126,961 rd   +  13,070 wr)
LLd misses:      121,744,698  (121,731,668 rd   +  13,030 wr)
D1  miss rate:          14.0% (       14.0%     +     2.9%  )
LLd miss rate:          12.5% (       12.5%     +     2.9%  )
LL refs:         136,141,454  (136,128,384 rd   +  13,070 wr)
LL misses:       121,746,052  (121,733,022 rd   +  13,030 wr)
LL miss rate:            4.3% (        4.3%     +     2.9%  )
```

Hardware: Stack/Heap Allocation

What are the practical differences between the stack and the heap?

- stack exhaustion
- heap fragmentation
- allocation speed
- overrun protection

Hardware: Stack/Heap Allocation

```
#include <stdlib.h>

int
main(void)
{
        int b, i, j, *buf, *a = &b;
        for (b = i = 0; i < 1000000; i++) {
                buf = malloc(4096 * sizeof(int));
                for (j = 0; j < 4096; j++)
                        buf[j] = (*a)++;
                free(buf);
        }
        return(EXIT_SUCCESS);
}
```

Hardware: Stack/Heap Allocation

```
#include <stdlib.h>

void
f(int *a)
{
        int i, buf[4096];
        for (i = 0; i < 4096; i++)
                buf[i] = (*a)++;
}

int
main(void)
{
        int a, i;
        for (a = i = 0; i < 1000000; i++)
                f(&a);
        return(EXIT_SUCCESS);
}
```

Hardware: Stack/Heap Allocation

```
% cc -g -W -Wall -o example6 example6.c
% cc -g -W -Wall -o example5 example5.c
% time ./example5
    0m22.42s real
    0m22.26s user
    0m0.00s system
% time ./example6
    0m27.27s real
    0m27.24s user
    0m0.00s system
```

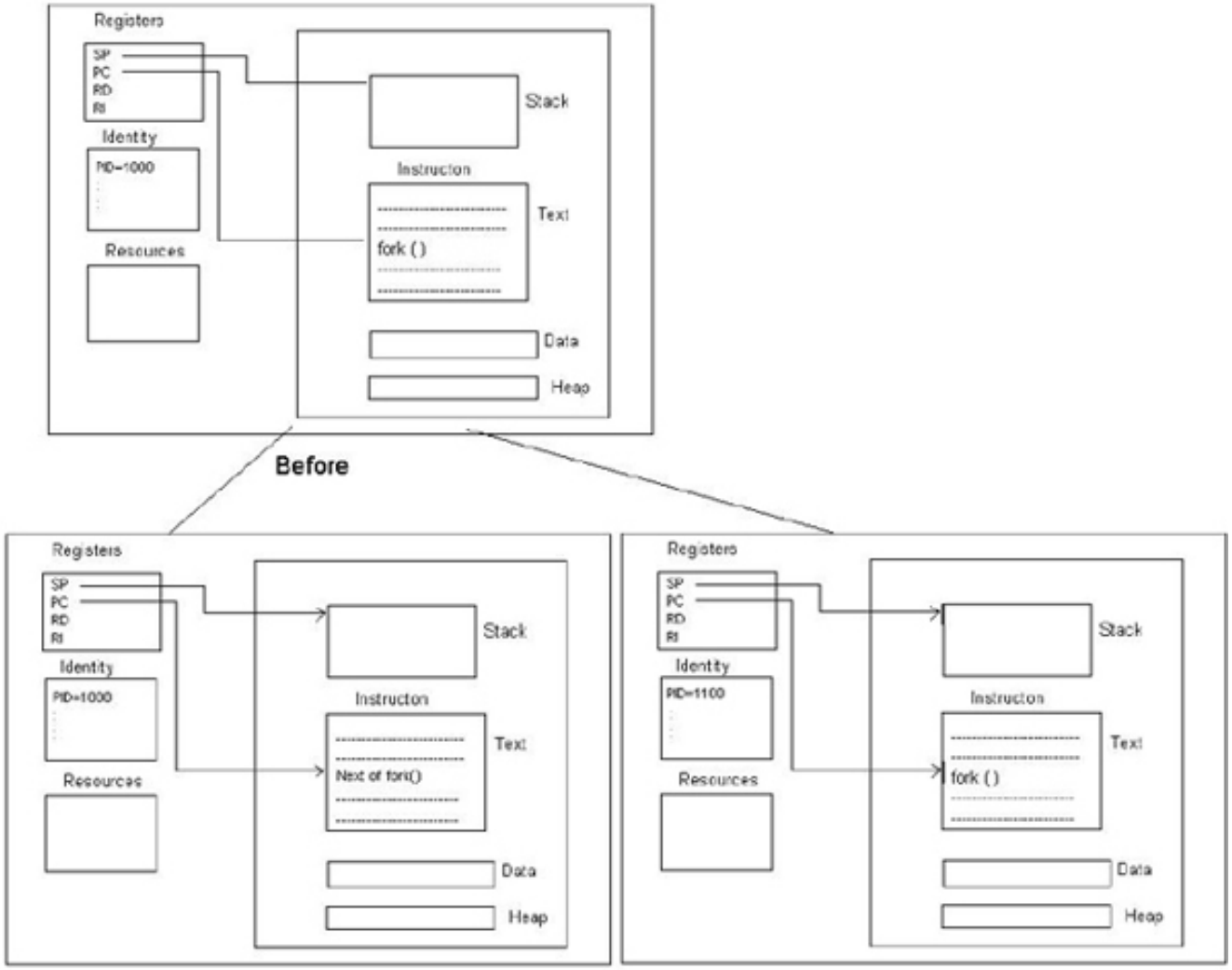
Shared Memory: Introduction

Consider the dot product:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

In the UNIX and C course, we examined a prime-test using `fork()` ("unshared memory") to create parallel processes. Can we do the same here?

Shared Memory: Fork Model



Shared Memory: Case Study

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

static float result(void);
static void f(int, float *, float *, int, int);

int
main(void)
{
        float *v1 = NULL, *v2 = NULL, in1, in2;
        size_t sz = 0;

        while (2 == scanf("%f,%f", &in1, &in2)) {
                v1 = realloc(v1, (sz + 1) * sizeof(float));
                v2 = realloc(v2, (sz + 1) * sizeof(float));
                v1[sz] = in1;
                v2[sz++] = in2;
        }

        if (0 == fork())
                f(0, v1, v2, 0, sz / 2);
        if (0 == fork())
                f(1, v1, v2, sz / 2, sz);

        wait(NULL); wait(NULL);
        printf("%g\n", result());
        free(v1); free(v2);
        return(EXIT_SUCCESS);
}
```

Shared Memory: Case Study

```
static float
result(void)
{
        FILE *f;
        float in1, in2;

        f = fopen("example3.0.dat", "r");
        fscanf(f, "%f", &in1);
        fclose(f);

        f = fopen("example3.1.dat", "r");
        fscanf(f, "%f", &in2);
        fclose(f);

        return(in1 + in2);
}
```

Shared Memory: Case Study

```
static void
f(int id, float *v1, float *v2, int start, int sz)
{
        FILE *f;
        char buf[64];
        int i;
        float res;

        for (res = 0.0, i = start; i < sz; i++)
                res += v1[i] * v2[i];

        snprintf(buf, sizeof(buf), "example3.%d.dat", id);
        f = fopen(buf, "w");
        fprintf(f, "%g\n", res);
        fclose(f);

        free(v1);
        free(v2);
        exit(EXIT_SUCCESS);
}
```

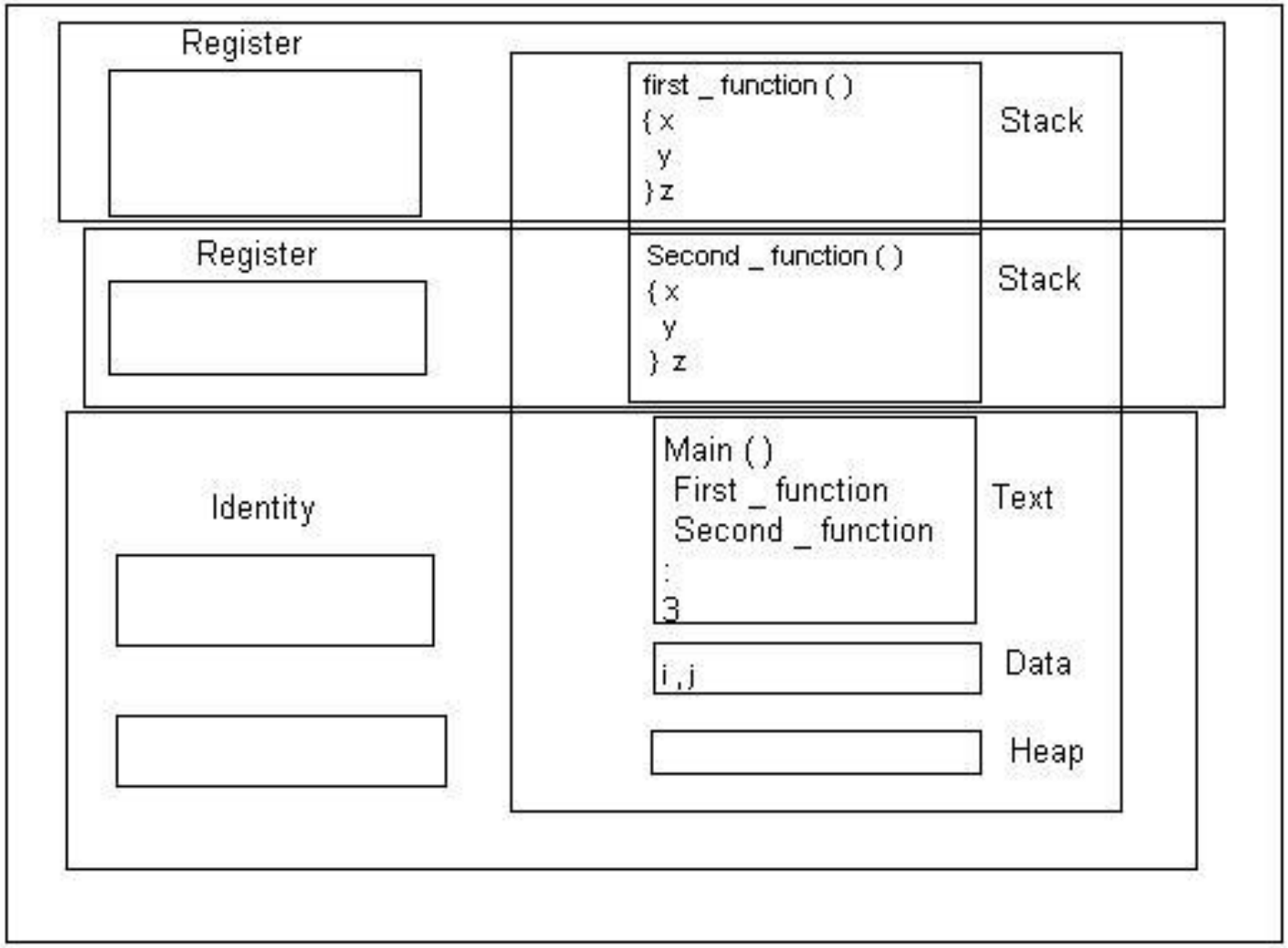
Shared Memory: Problem

Instead, can we allow multiple processes to share a region of memory? Consider using threads ("light-weight processes").

- Sharing data section instead of cloning.
- Faster cleanup/teardown.
- Manual synchronisation of memory.

The UNIX standard for threading is *pthreads*, POSIX threads.

Shared Memory: Thread Model



Shared Memory: Case Study

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static float *v1, *v2;
static size_t sz;

static void *
f(void *arg)
{
        int i, start, end;
        float res;

        start = *(float *)arg > 0.0 ? 0 : sz / 2;
        end = *(float *)arg > 0.0 ? sz / 2 : sz;

        for (res = 0.0, i = start; i < end; i++)
                res += v1[i] * v2[i];

        *(float *)arg = res;
        return(NULL);
}
```

Shared Memory: Case Study

```
int
main(void)
{
        float in1, in2;
        pthread_t p1, p2;

        while (2 == scanf("%f,%f", &in1, &in2)) {
                v1 = realloc(v1, (sz + 1) * sizeof(float));
                v2 = realloc(v2, (sz + 1) * sizeof(float));
                v1[sz] = in1;
                v2[sz++] = in2;
        }

        in1 = -1.0;
        pthread_create(&p1, NULL, f, &in1);
        in2 = 1.0;
        pthread_create(&p2, NULL, f, &in2);

        pthread_join(p1, NULL);
        pthread_join(p2, NULL);

        printf("%g\n", in1 + in2);

        free(v1);
        free(v2);
        return(EXIT_SUCCESS);
}
```

Shared Memory: Case Study

The code is simpler and scales better. Now consider a simple benchmark:

```
% cc -O2 -g -W -Wall -o example4 example4.c -lpthread
% i=0 ; while [ $i -lt 90000 ] ; do \
> echo $i,$i ; i=$(($i+1)) ; done | time ./example4
    0m1.20s real
    0m0.17s user
    0m0.57s system
% cc -O2 -g -W -Wall -o example3 example3.c
% i=0 ; while [ $i -lt 90000 ] ; do \
> echo $i,$i ; i=$(($i+1)) ; done | time ./example3
    0m1.52s real
    0m0.15s user
    0m0.56s system
```