Concurrency: Threads

CS 537: Introduction to Operating Systems

Louis Oliphant

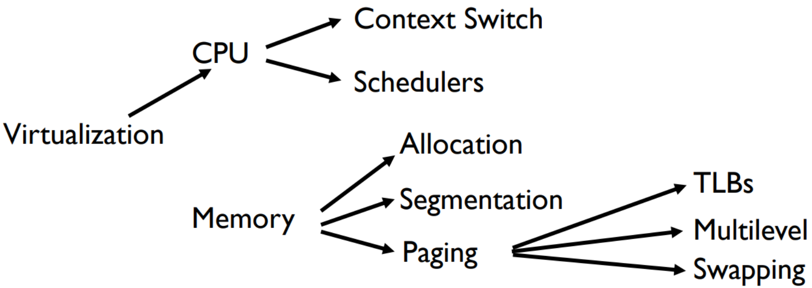
University of Wisconsin - Madison

Fall 2024

Administrivia

- Code reviews begin this Friday (look for an email from your TA)
- Midterm 1 Scheduled:
  - Regular Time: Oct 15th, 5:45-7:15
    - Last name A-K: Van Vleck B102
    - Last name L-Z: Ingraham B10
  - Unable to attend? Fill out this form
  - Alternate Time: Oct 16th, 5:45-7:15, Psych Bldg 121
  - McBurney Time: Oct 15th, 5:45-8:45pm, CS 1257
  - Bring **#2 Pencil**, one sheet of **8.5x11 (A4) notes**, and **UW Student ID**

Review: Virtualization



Correction: Bounds Check with Negative Growth

Segment	Base	Size (max 4K)	Grows Positive?
$Code_{00}$	32K	2K	1
$Heap_{01}$	34K	3K	1
$Stack_{11}$	28K	2K	0

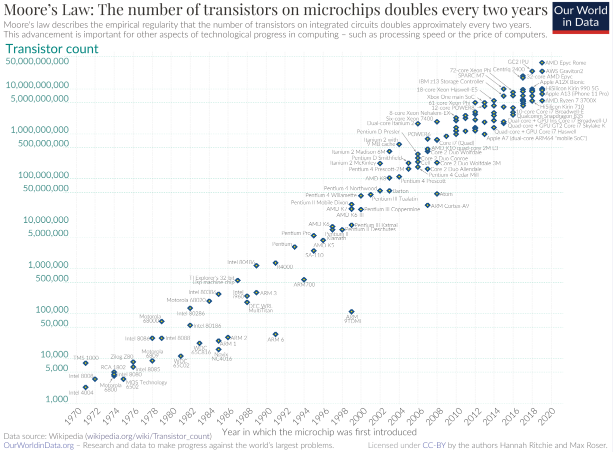
Figure 16.4: **Segment Registers (With Negative-Growth Support)**

Virtual address 15KB, which should map to physical address 27KB. Binary form is: `11 1100 0000 0000` (hex 0x3C00). Hardware uses top two bits (11) for segment, left with offset of 3KB. For the negative offset subtract maximum segment size from 3KB: **3KB minus 4KB which equals -1KB**. Add negative offset (-1KB) to the base (28KB) to get physical address: 27KB. **The bounds check can be calculated by ensuring the absolute value of the negative offset is less than or equal to the segment's current size (in this case, 2KB)**.

Concurrency: Motivation

CPU Performance

Goal: Write applications that fully **utilize many cores**



Option 1: Communicating Processes

- Build Application using multiple processes
  - Example: Google Chrome (each tab is a process)
  - Communicate via `pipe()` or something similar
- Pros
  - Don't need new abstraction
  - Good for security
- Cons
  - Cumbersome programming
  - High communication overheads
  - Expensive context switch

Option 2: Threading

- New abstraction: **thread**
- Threads are like processes, except:
  - Multiple threads of same process **share an address space**
- Divide large task across several cooperative threads
- Communicate through **shared address space**

Common Concurrency Programming Models

- **Embarrassingly parallel**
  - e.g., ray tracing, database select, compiling separate files
- **Producer/Consumer**
  - Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
- **Pipeline**
  - Task is divided into series of subtasks, each processed on a different core
- **Defer work with background thread**
  - One thread performs non-critical work in the background (when CPU would be idle)

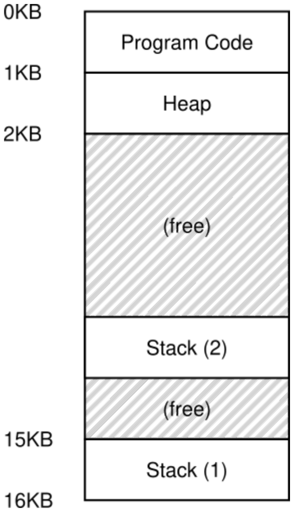
Thread vs. Process

Multiple threads share:

- Process ID (PID)
- Address space: Code (instructions), Most data (heap)
- Open file descriptors
- Current working directory
- User and group ID

Each thread has its own:

- Thread ID (TID)
- Set of registers, including PC and SP
- Stack for local vars and return address



```
void *mythread(void *arg) {
  printf("%s\n", (char *) arg);
  return NULL;
}

int main() {
  pthread_t p1, p2;
  printf("begin\n");
  Pthread_create(&p1, NULL, mythread, "A");
  Pthread_create(&p2, NULL, mythread, "B");
  // join waits for threads to finish
  Pthread_join(p1, NULL);
  Pthread_join(p2, NULL);
  printf("end\n");
  return 0;
}
```

Example Thread Trace 1 On Uniprocessor

main	**Thread 1**	**Thread 2**
start		
print "begin"		
create Thread 1		
create Thread 2		
wait for T1		
	print "A"	
wait for T2		
		print "B"
print "end"		

Example Thread Trace 2 On Uniprocessor

main	**Thread 1**	**Thread 2**
start		
print "begin"		
create Thread 1		
	print "A"	
create Thread 2		
		print "B"
wait for T1 *(returns immediately)*		
wait for T2 *(returns immediately)*		
print "end"		

Example Thread Trace 3 On Uniprocessor

main	Thread 1	Thread 2
start		
print "begin"		
create Thread 1		
create Thread 2		
		print "B"
wait for T1		
	print "A"	
wait for T2 *(returns immediately)*		
print "end"		

A Program's Behavior is *non-deterministic*.

Example Sharing Data

```c
int max;
volatile int counter = 0; // shared global variable

void *mythread(void *arg) {
    char *letter = arg;
    int i; // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);
    for (i = 0; i < max; i++) {
        counter = counter + 1; // shared: only one
    }
    printf("%s: done\n", letter);
    return NULL;
}
```

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: main-first <loopcount>\n");
        exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: begin [counter = %d] [%x]\n", counter,
           (unsigned int) &counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [counter: %d]\n [should: %d]\n",
           counter, max*2);
    return 0;
}
```

```
prompt> ./threads 1000000
main: begin [counter = 0] [2df4a030]
A: begin [addr of i: 0x7fc6debfee3c]
B: begin [addr of i: 0x7fc6de3fde3c]
A: done
B: done
main: done
 [counter: 1013349]
 [should: 2000000]
```

Uncontrolled Scheduling – **Race Condition**

```
counter=counter+1;  // Critical Section
```

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

			(after instruction)		
OS	**Thread 1**	**Thread 2**	**PC**	**eax**	**counter**
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

What value is counter? Starting value = 50

```
Thread 1              Thread 2
mov 0x8049a1c, %eax
add $0x1, %eax
                      mov 0x8049a1c, %eax
mov %eax, 0x8049a1c
                      add $0x1, %eax
                      mov %eax, 0x8049a1c
```

```
Thread 1              Thread 2
                      mov 0x8049a1c, %eax
                      add $0x1, %eax
                      mov %eax, 0x8049a1c
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

```
Thread 1              Thread 2
                      mov 0x8049a1c, %eax
mov 0x8049a1c, %eax
                      add $0x1, %eax
add $0x1, %eax
                      mov %eax, 0x8049a1c
mov %eax, 0x8049a1c
```

What value is counter? Starting value = 50

```
Thread 1              Thread 2
mov 0x8049a1c, %eax
add $0x1, %eax
                      mov 0x8049a1c, %eax
mov %eax, 0x8049a1c
                      add $0x1, %eax
                      mov %eax, 0x8049a1c        counter = 51
```

```
Thread 1              Thread 2
                      mov 0x8049a1c, %eax
                      add $0x1, %eax
                      mov %eax, 0x8049a1c
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c                               counter = 52
```

```
Thread 1              Thread 2
                      mov 0x8049a1c, %eax
mov 0x8049a1c, %eax
                      add $0x1, %eax
add $0x1, %eax
                      mov %eax, 0x8049a1c
mov %eax, 0x8049a1c                               counter = 51
```

Non-Determinism

Concurrency leads to **non-deterministic** results

- Different results even with same inputs
- **Race Condition** – results depend upon the scheduling order

Whether bug manifests depends on CPU scheduling!

What We Want

Want 3 instructions to execute as an uninterruptable group

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Want them to be **Atomic** – "as a unit" or "all or nothing". The three instructions should all run together (or not at all).

Synchronization Primitives

- Hardware support helps to build **Synchronization primitives**:
  - **Lock**
  - **Condition Variable**
  - **Semaphore**
  - **Barrier**
- Used to create atomicity for critical sections
- Also used to make one thread wait for another thread to complete some action before continuing

Why in OS Class?

- OS is the first concurrent program
- Page tables, process lists, file system structures, and most kernel data must be accessed using proper synchronization primitives.

Thread Creation

```c
#include <pthread.h>

typedef struct __myarg_t {
   int a;
   int b;
} myarg_t;

void *mythread(void *arg) {
   myarg_t *m = (myarg_t *) arg;
   printf("%d %d\n", m->a, m->b);
   return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p;
  myarg_t args;
  args.a = 10;
  args.b = 20;
  rc = pthread_create(&p, NULL, mythread, &args); //success returns 0
}
```

Thread Joining (and returning values)

```
typedef struct __myret_t {
  int x;
  int y;
} myret_t;

void *mythread(void *arg) {
  ...
  myret_t *r = Malloc(sizeof(myret_t));
  r->x = 1;
  r->y = 2;
  return (void *) r;
}

int main(int argc, char *argv[]) {

  myret_t *m;
  ...
  rc=pthread_create(...);
  rc=pthread_join(p, (void **) &m); //success returns 0
  printf("returned %d %d\n", m->x, m->y);
  free(m);
}
```