Concurrency: Threads

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2023

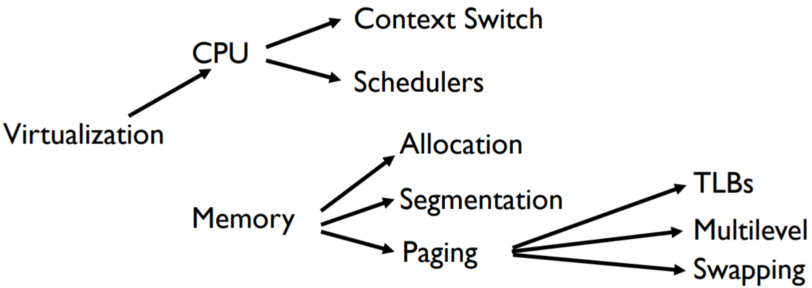
Administrivia

- Project 2 Grading
  - Questions / Corrections? email Danial (saleem5@wisc.edu) and Aditya (adassarma@cs.wisc.edu)
- Project 3 Grading
  - Very Challenging Project (especially job control)
  - Extra Credit Possible
  - Fork()/Exec()/Wait() 45 pts
- Project 4 due Oct 24th @ 11:59pm
  - Discussion section cover:
    - P4 concepts
    - Steps you should follow to find your own bugs (before asking for help)

Administrivia (cont.)

- Exam 1
  - Mean: 74.1% Max: 93.3% Min: 43.3% Std: 5.87 points
  - Will release solution and grades after Epic takes exam
- Survey (10-18 thru 10-24)
  - You should receive an email inviting you to provide feedback about your course learning experience.
  - Please provide constructive feedback

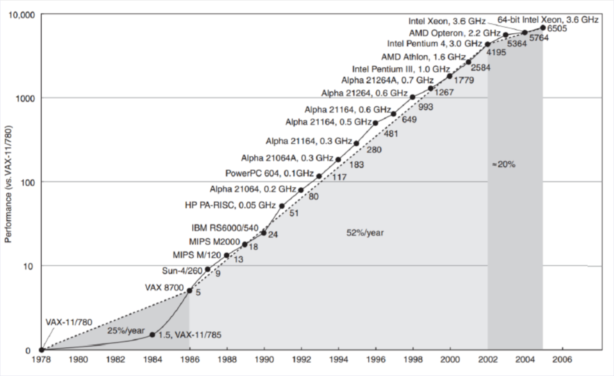
Review: Virtualization



Concurrency: Motivation

CPU Performance

- CPU Trend: Multiple cores – each same speed
- 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 knew abstraction
  - Good for security
- Cons
  - Cumbersome programming
  - High communication overheads
  - Expensive context switch

Option 2: Threading

- New abstraction: **thread**
- Threads 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 Programming Models

Multi-threaded progams tend to be structured as:

- **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 of which is handled **in series** by a different thread
- **Defer work with background thread**
  - One thread performs non-critical work in the background (when CPU would be idle)

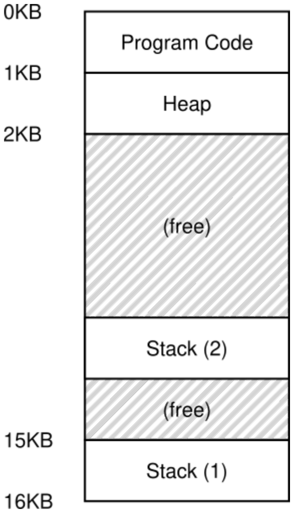
Thread vs. Process

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



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

int main(int argc, char *argv[]) {
    if (argc != 1) {
        fprintf(stderr, "usage: main\n");
        exit(1);
    }

    pthread_t p1, p2;
    printf("main: begin\n");
    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: end\n");
    return 0;
}
```

Example Thread Trace 1

main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
		runs
		prints “B”
		returns
prints “main: end”		

Example Thread Trace 2

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Example Thread Trace 3

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

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 100000
B: begin
A: begin
A: done
B: done
main: done
 [counter: 1094044]
 [should:  2000000]
```

Uncontrolled Scheduling – **Race Condition**

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

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

OS	Thread 1	Thread 2	(after instruction) 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**:
  - **Monitor**
  - **Lock**
  - **Semaphore**
  - **Condition Variable**
- 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);
}
```