Operating Systems

Threads

ENCE 360

Outline

- Model
- Motivation
- Libraries

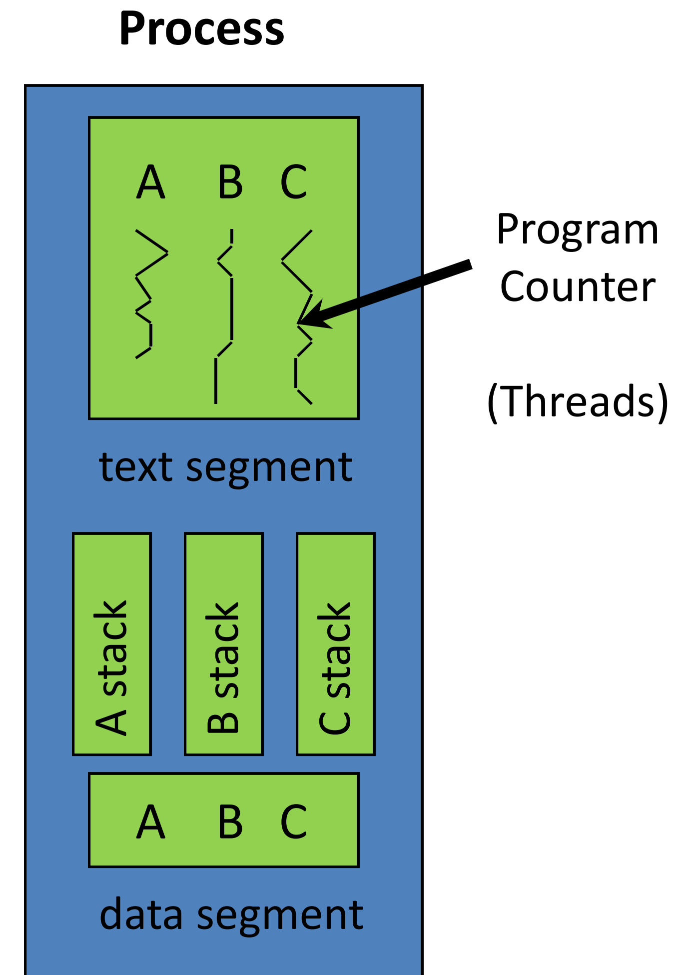
Chapter 2.2
MODERN OPERATING SYSTEMS (MOS)
By Andrew Tanenbaum

Chapter 26.1, 26.2
OPERATING SYSTEMS: THREE EASY PIECES
By Arpaci-Dusseau and Arpaci-Dusseau

Threads (Lightweight Processes)

- Single sequence of execution within a process
  - Basic unit of CPU utilization
- Private
  - Program counter
  - Register set
  - Stack space
- Shared
  - Code section
  - Data section
  - OS resources

Because have some process properties (but not all), often called lightweight process



Thread – Private vs. Shared

```
        int g_x;
        B() {
          int x = 10;
A PC →    printf(x);
        }
        A(int x) {
B PC →    B();
        }
        main() {
          A(1);
        }
```

Assume two threads (A and B) in same process running code on left

What is shared between them?

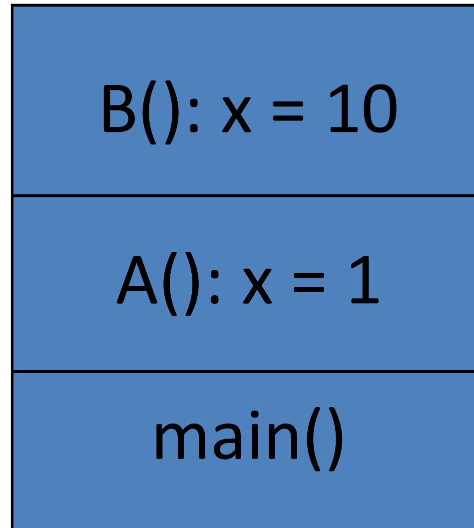
What is private?

Hint: remember other components of system, too!

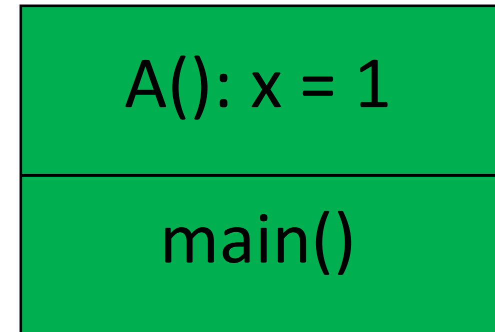
Thread – Private vs. Shared

```
int g_x;
B() {
  int x = 10;
  printf(x);      <- A PC
}
A(int x) {
  B();            <- B PC
}
main() {
  A(1);
}
```

g_x



Thread A

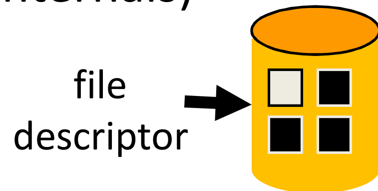


Thread B

Shared

Private

(OS Internals)



includes `stdio`, `stdin`

> Beware non-thread safe library/system calls!
> e.g., `strtok()`, `rand()`, `readdr()`
> Use thread-safe version: e.g., `rand_r()`

SOS: "`pcb-thread.h`"

Thread – Private vs. Shared Summary

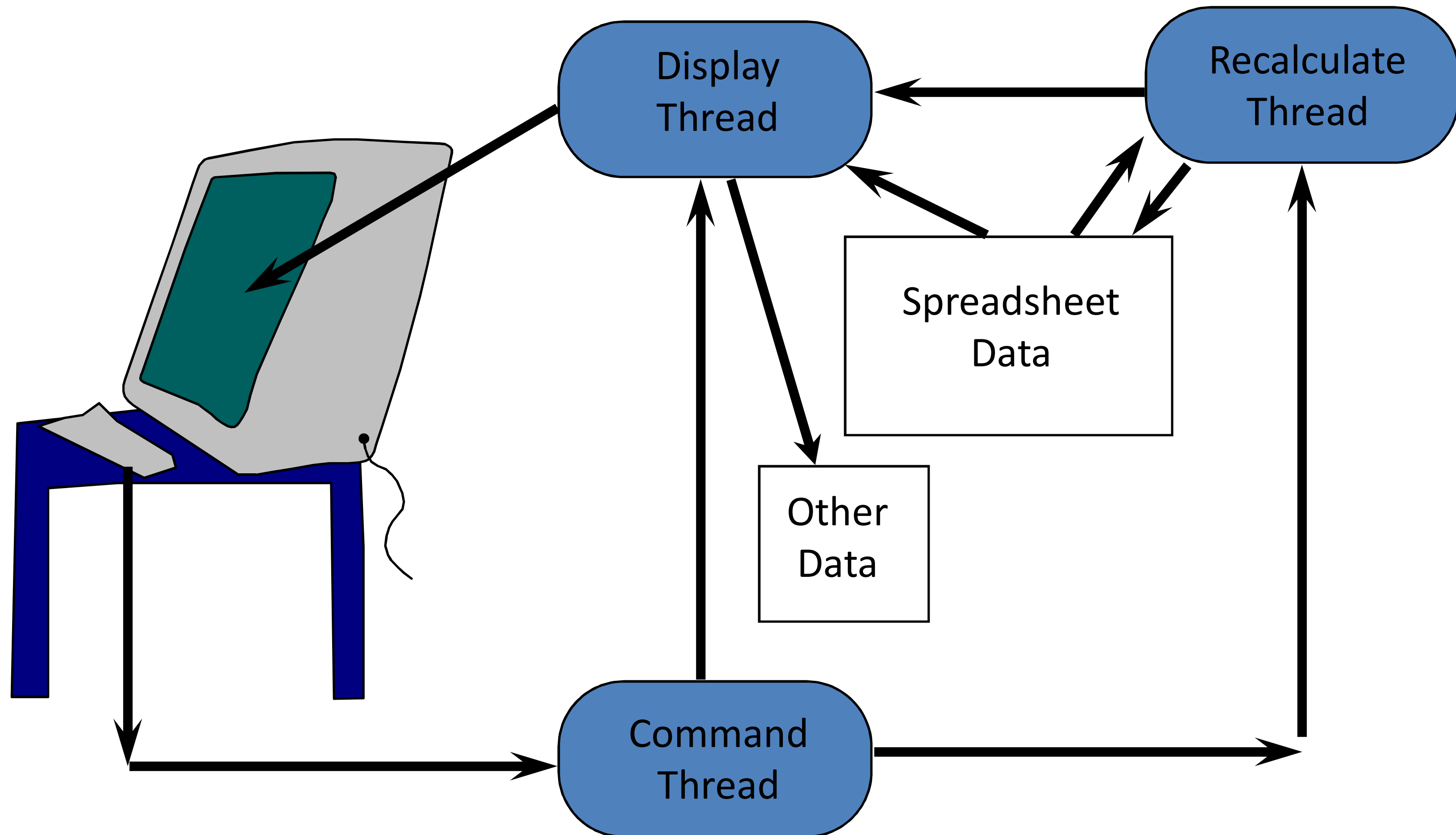
Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

(Shared by each thread) (Private to each thread)

Outline

- Model (done)
- Motivation (next)
- Libraries

Example: A Threaded Spreadsheet



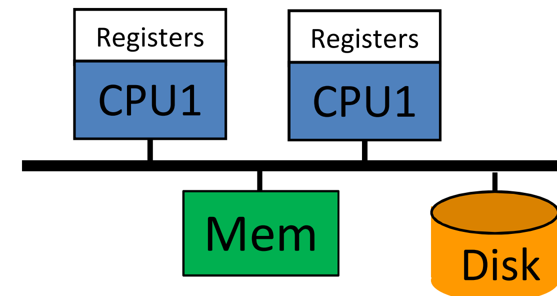
What Kinds of Programs to Thread?

What Kinds of Programs to Thread?

- Independent tasks (e.g., spreadsheet)
- Single program, concurrent operation
  - Servers: e.g., file server, Web server
  - OS kernels: concurrent system requests by multiple processes/users

- Especially when block for I/O!
  → With threads, can continue execution in another thread

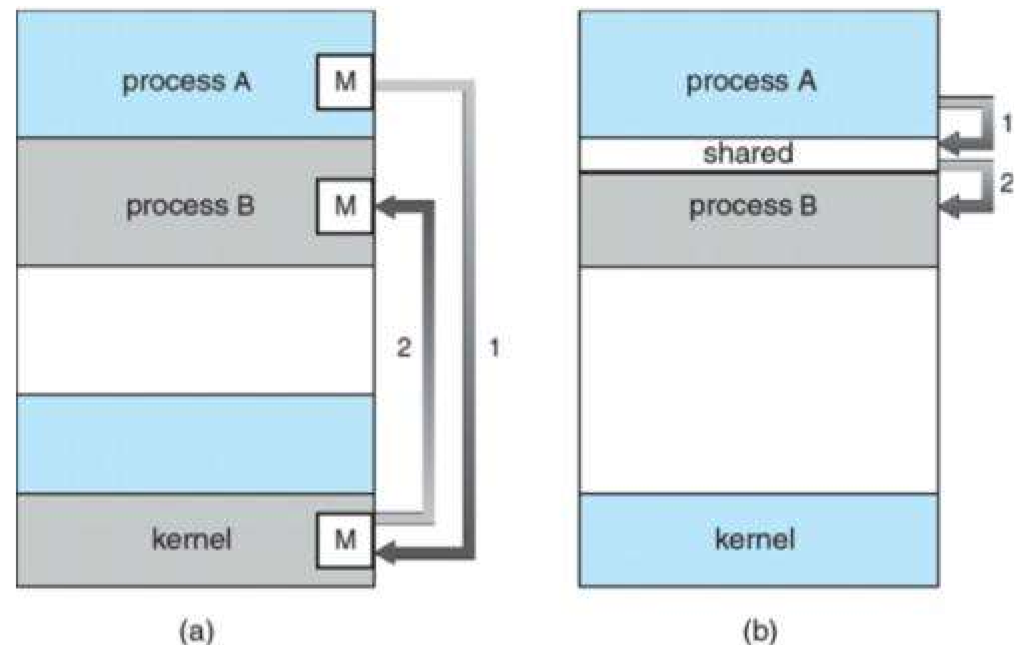
- Especially with multiple-CPUs!
  → Each CPU can run one thread



Potential Thread Benefits

"What about just using multiple communicating processes?"
Sure, this can be made to work

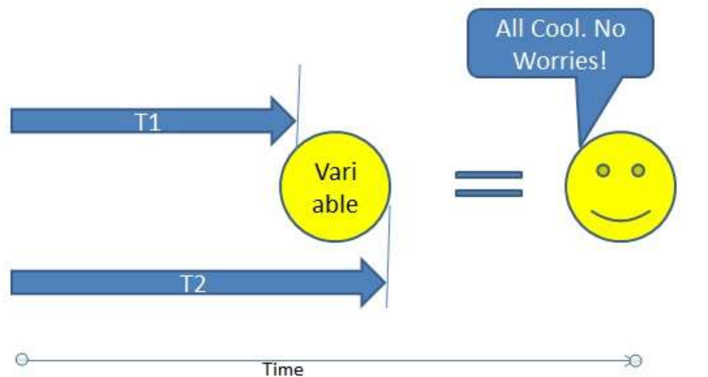
- But separate code needed to coordinate processes
  a) e.g., pipes
  b) e.g., shared memory + locks



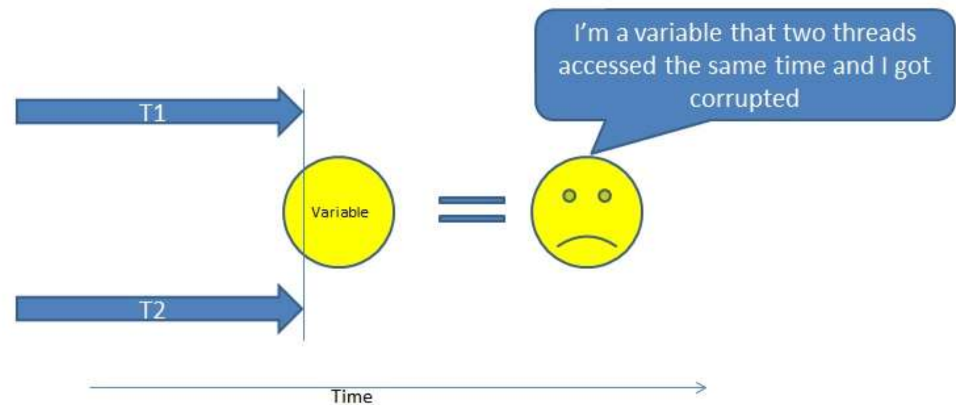
- And debugging tougher
- Also, processes "cost" more
  - Up to 100x longer to create/destroy
  - Far more memory (since not shared)
  - Slower to context switch among

- Few thousand processes not ok
- Few thousand threads ok

Warning Using Threads



Two threads accessing the same variable at different times



Two threads accessing the same variable at the same time

- Versus single threaded program, can be more difficult to write and debug code
- Concurrency problems for shared resources
  - Global variables
  - But also system calls and system resources
- *Only* use threads when performance an issue (blocking too costly and/or multi-processor is available)
- So … *is* performance an issue?

Is Performance an Issue?

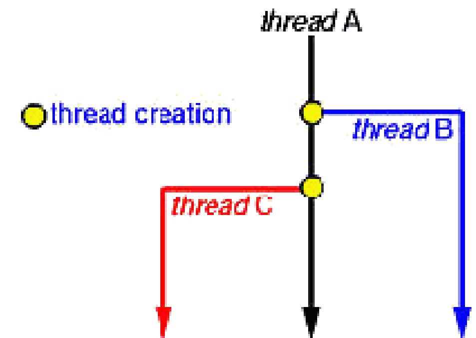
- You don’t need to improve performance of your code
- Most important → Code that works, is robust
- More important → Code that is clear, readable
  - It *will* be re-factored
  - It *will* be modified/extended by others (even you!)
- Less important → Code that is efficient, fast
  - Is performance really issue?
  - Can hardware upgrade fix performance problems?
    - *e.g., Moore’s Law* (http://en.wikipedia.org/wiki/Moore's_law)
  - Can design fix performance problems?
- Ok, so you *do* really need to improve performance
  - Use threads … but carefully! (Concurrency)

Outline

- Model (done)
- Motivation (done)
- Libraries (next)

Thread Libraries for C/C++

- Dozens: https://en.wikipedia.org/wiki/thread-lib
- Main – POSIX threads (pthreads) and Windows
  - Totally different
- Fortunately, common functionality
  - Create, Destroy, Join, Yield
  - Lock/Unlock (for concurrency)




```
#include <pthread.h>
Linker: -lpthread
```

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

POSIX Threads - Example

See: “threads-hello.c”

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

// Do some work.
void *worker(void *arg) {
  printf("This is a thread.  Hello, world!\n");
  pthread_exit(NULL);
}

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

  pthread_t p1, p2;
  if (pthread_create(&p1, NULL, worker, NULL) != 0) {
    printf("Error! p1: pthread_create failed.");
    return 0;
  }
  if (pthread_create(&p2, NULL, worker, NULL) != 0) {
    printf("Error! p2: pthread_create failed.");
    return 0;
  }

  printf("Main thread waiting for children to finish....\n");
  pthread_join(p1, NULL);
  pthread_join(p2, NULL);

  printf("Children finished.  Exiting.\n");
  return 0;
}
```

Example – Thread vs. Fork (1 of 2)

```
// fork.c

int global = 5;

int main(int argc, char *argv[]) {
  int local = 10;

  printf("Start: global %d, local %d\n", global, local);
  int pid = fork();
  if (pid < 0) {    /* Fail. */
    perror("Fork failed.");
    return -1;
  }
  if (pid == 0) {   /* Child. */
    printf(" Child.  My id is %d.\n", getpid());
    global++;
    local++;
  } else {          /* Parent. */
    printf("Parent.  My child id is %d.\n", pid);
    global--;
    local--;
  }
  printf("  End: global %d, local %d\n", global, local);
}
```

See: "`fork.c`"

What do you think the output will be?

Example – Thread vs. Fork (2 of 2)

"thread.c"

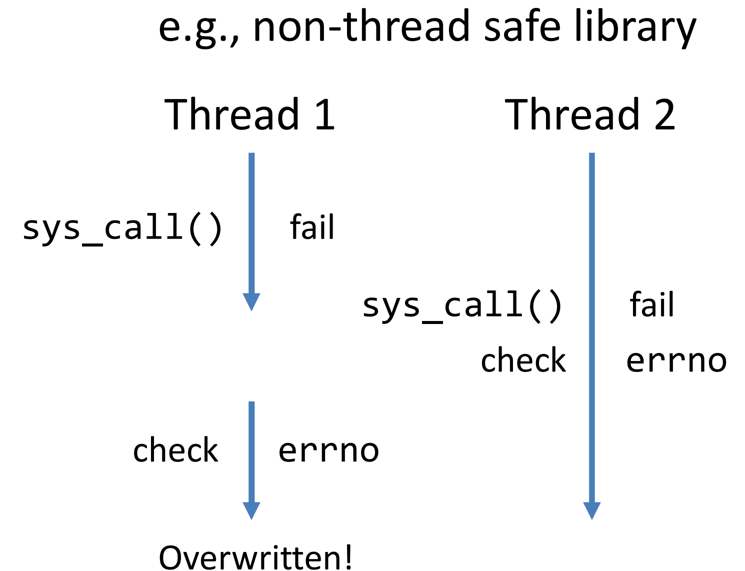
```
// thread.c

volatile int global = 5;

void *worker() {
  int local = 10;
  printf("Child thread.\n");
  local++;
  global++;
  pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
  pthread_t p;
  int local = 10;

  printf("Start: global %d, local %d\n", global, local);
  if (pthread_create(&p, NULL, worker, NULL) != 0) {
    printf("Error! pthread_create failed.");
    return 0;
  } else {
    printf("Parent thread.\n");
    sleep(2);
    global--;
    local--;
  }
  printf("  End: global %d, local %d\n", global, local);
}
```

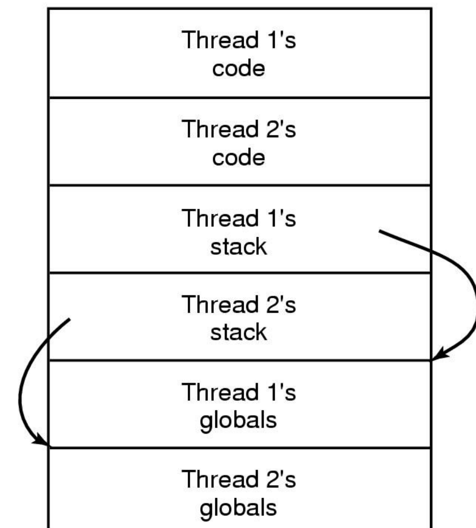
What do you think the output will be?

Making Single-Threaded Code Multithreaded

- Many legacy systems single-threaded
- If benefit, (see "performance?" above) can convert → But tricky!
- Yes, local variables easy
- Many library functions expect to be single-threaded
  - Not re-entrant code
  - Look for _r versions (e.g., strtok_r())
- And global variables difficult
  - Can create private "globals"
- Still other issues, signal handling, stack management, and so on

→ Proceed with caution!





Outline

- Model (done)
- Motivation (done)
- Libraries (done)