# threads

# why threads?

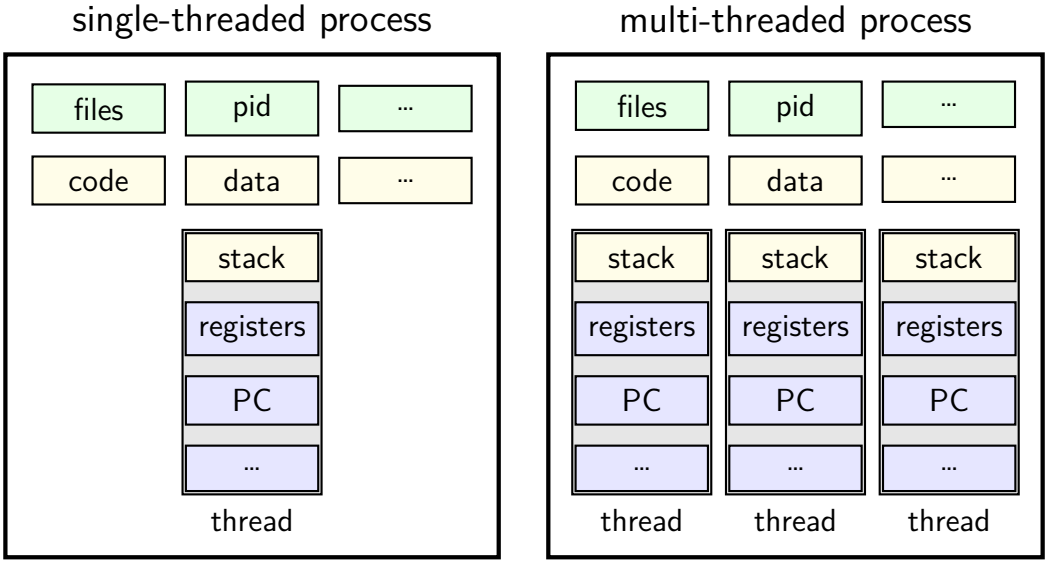
concurrency: different things happening at once

- one thread per user of web server?
- one thread per page in web browser?
- one thread to play audio, one to read keyboard, ...?
- ...

parallelism: do same thing with more resources

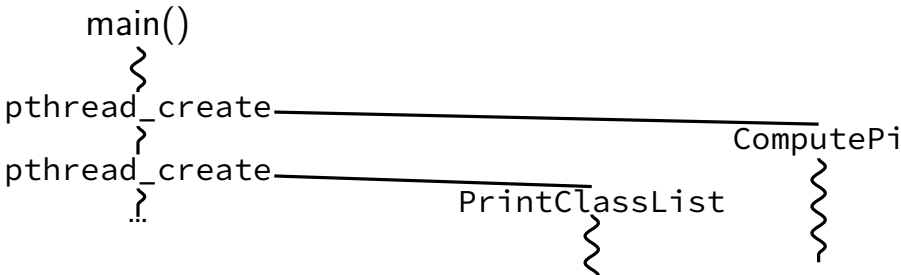
- multiple processors to speed-up simulation (life assignment)

# single and multithread processes



# pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
  pthread_t pi_thread, list_thread;
  if (0 != pthread_create(&pi_thread, NULL, ComputePi, NULL))
      handle_error();
  if (0 != pthread_create(&list_thread, NULL, PrintClassList, NULL))
      handle_error();
  ... /* more code */
}
```



# pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
  pthread_t pi_thread, list_thread;
  if (0 != pthread_create(&pi_thread, NULL, ComputePi, NULL))
      handle_error();
  if (0 != pthread_create(&list_thread, NULL, PrintClassList, NULL))
      handle_error();
  ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

# pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
  pthread_t pi_thread, list_thread;
  if (0 != pthread_create(&pi_thread, NULL, ComputePi, NULL))
      handle_error();
  if (0 != pthread_create(&list_thread, NULL, PrintClassList, NULL))
      handle_error();
  ... /* more code */
}
```

pthread_create arguments:

*thread identifier*

function to run thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

# pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
  pthread_t pi_thread, list_thread;
  if (0 != pthread_create(&pi_thread, NULL, ComputePi, NULL))
      handle_error();
  if (0 != pthread_create(&list_thread, NULL, PrintClassList, NULL))
      handle_error();
  ... /* more code */
}
```

pthread_create arguments:

thread identifier

*function to run* thread starts here, terminates if this function returns

thread attributes (extra settings) and function argument

# pthread_create

```
void *ComputePi(void *argument) { ...  }
void *PrintClassList(void *argument) { ...  }
int main() {
  pthread_t pi_thread, list_thread;
  if (0 != pthread_create(&pi_thread, NULL, ComputePi, NULL))
      handle_error();
  if (0 != pthread_create(&list_thread, NULL, PrintClassList, NULL))
      handle_error();
  ... /* more code */
}
```

pthread_create arguments:

thread identifier

function to run thread starts here, terminates if this function returns

*thread attributes (extra settings) and function argument*

# a threading race

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In␣the␣thread\n");
    return NULL;
}
int main() {
    printf("About␣to␣start␣thread\n");
    pthread_t the_thread;
    /* assume does not fail */
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done␣starting␣thread\n");
    return 0;
}
```

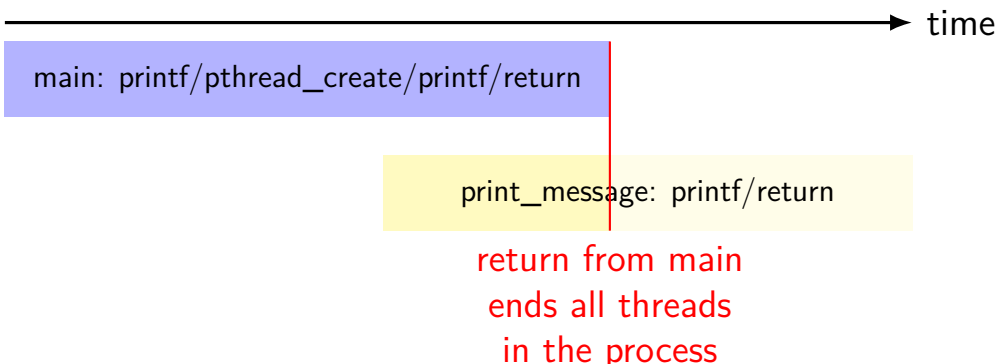
My machine: outputs `In the thread` *about 4% of the time*.

# a race

returning from main *exits the entire process* (all its threads)

same as calling exit; not like other threads

race: main's return 0 or print_message's printf first?



# fixing the race (version 1)

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In␣the␣thread\n");
    return NULL;
}
int main() {
    printf("About␣to␣start␣thread\n");
    pthread_t the_thread;
    /* missing: error checking */
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done␣starting␣thread\n");
    pthread_join(the_thread, NULL);  /* WAIT FOR THREAD */
    return 0;
}
```

# fixing the race (version 2; not recommended)

```
#include <pthread.h>
#include <stdio.h>
void *print_message(void *ignored_argument) {
    printf("In␣the␣thread\n");
    return NULL;
}
int main() {
    printf("About␣to␣start␣thread\n");
    pthread_t the_thread;
    /* missing: error checking */
    pthread_create(&the_thread, NULL, print_message, NULL);
    printf("Done␣starting␣thread\n");
    pthread_exit(NULL);
}
```

# pthread_join, pthread_exit

R = `pthread_join`(X, &P): wait for thread X, copies return value into P

- like `waitpid`, but for a thread
- thread return value is pointer to anything
- R = 0 if successful, error code otherwise

`pthread_exit`: exit current thread, returning a value

- like `exit` or returning from main, but for a single thread
- same effect as returning from function passed to pthread_create

# a note on error checking

from pthread_create manpage:

**ERRORS**

**EAGAIN** Insufficient resources to create another thread, or a system-imposed limit on the number of threads was encountered. The latter case may occur in two ways: the **RLIMIT_NPROC** soft resource limit (set via **setrlimit**(2)), which limits the number of process for a real user ID, was reached; or the kernel's system-wide limit on the number of threads, /proc/sys/kernel/threads-max, was reached.

**EINVAL** Invalid settings in attr.

**EPERM** No permission to set the scheduling policy and parameters specified in attr.

special constants for *return value*

same pattern for many other pthreads functions

pthread_join, pthread_mutex_...(later), ...

will often omit error checking in slides for brevity

# error checking pthread_create

```
int error = pthread_create(...);
if (error != 0) {
    /* print some error message */
}
```

# sum example (only globals)

```
int values[1024];  int results[2];
void *sum_front(void *ignored_argument) {
    int sum = 0;
    for (int i = 0; i < 512; ++i) { sum += values[i]; }
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    /* missing: error handling */
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL); pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

# sum example (only globals)

values, results: global variables — shared

```
int values[1024];  int result
void *sum_front(void *ignored_argument) {
    int sum = 0;
    for (int i = 0; i < 512; ++i) { sum += values[i]; }
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    /* missing: error handling */
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL); pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

# sum example (only globals)

two different functions
happen to be the same except for some numbers

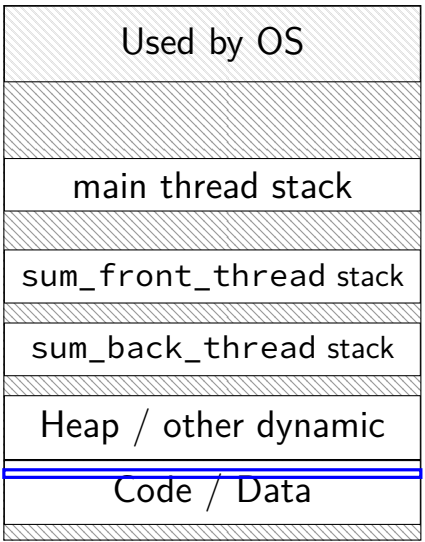
```
int values[1024];  int
void *sum_front(void
    int sum = 0;
    for (int i = 0; i < 512; ++i) { sum += values[i]; }
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    /* missing: error handling */
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL); pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

# sum

values returned from threads
via global array instead of return value
(partly to illustrate that memory is shared,
partly because this pattern works when we don't join (later))

```
int valu
void *su
    int
    for
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    return NULL;
}
int sum_all() {
    pthread_t sum_front_thread, sum_back_thread;
    /* missing: error handling */
    pthread_create(&sum_front_thread, NULL, sum_front, NULL);
    pthread_create(&sum_back_thread, NULL, sum_back, NULL);
    pthread_join(sum_front_thread, NULL); pthread_join(sum_back_thread, NULL);
    return results[0] + results[1];
}
```

# thread_sum memory layout



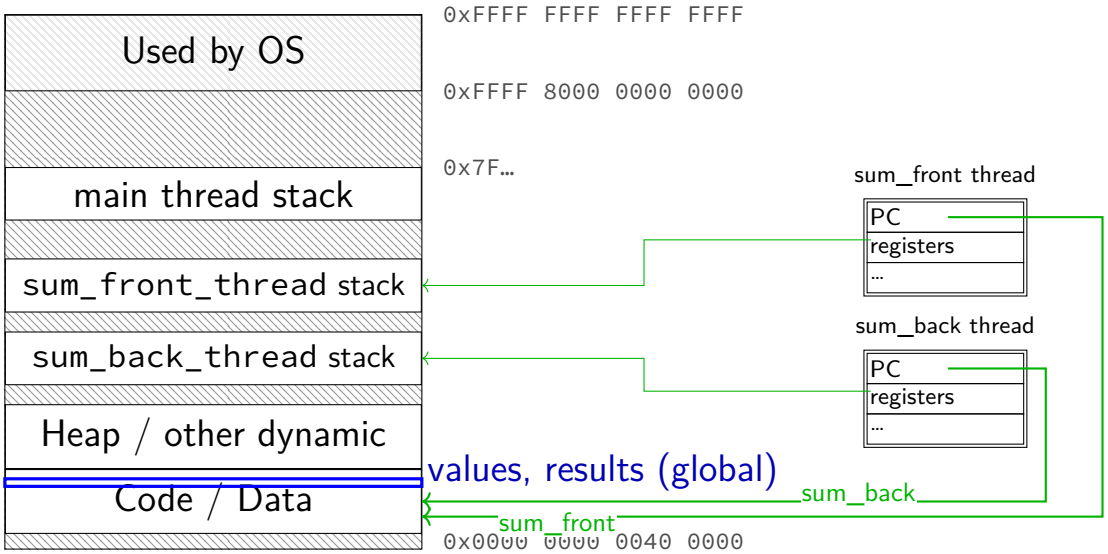

0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

0x0000 0000 0040 0000

# thread_sum memory layout



# sum example (to global, with thread IDs)

```
int values[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += values[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    /* missing: error handling */
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

# sum example (to global, with thread IDs)

values, results: global variables — shared

```
int values[1024];
int results[2];
void *sum_thread(void *argument) {
    int id = (int) argument;
    int sum = 0;
    for (int i = id * 512; i < (id + 1) * 512; ++i) {
        sum += values[i];
    }
    results[id] = sum;
    return NULL;
}
int sum_all() {
    /* missing: error handling */
    pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, sum_thread, (void *) i);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return results[0] + results[1];
}
```

# sum example (info struct)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    struct ThreadInfo *my_info =  (struct ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) { sum += values[i]; }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; struct ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i) { pthread_join(threads[i], NULL); }
    return info[0].result + info[1].result;
}
```

# sum example (info struct)

```
int values[1024];
struct ThreadInfo
    int start, end, result;
};
void *sum_thread(void *argument) {
    struct ThreadInfo *my_info =  (struct ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) { sum += values[i]; }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; struct ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i) { pthread_join(threads[i], NULL); }
    return info[0].result + info[1].result;
}
```

values: global variable — shared

# sum example (info struct)

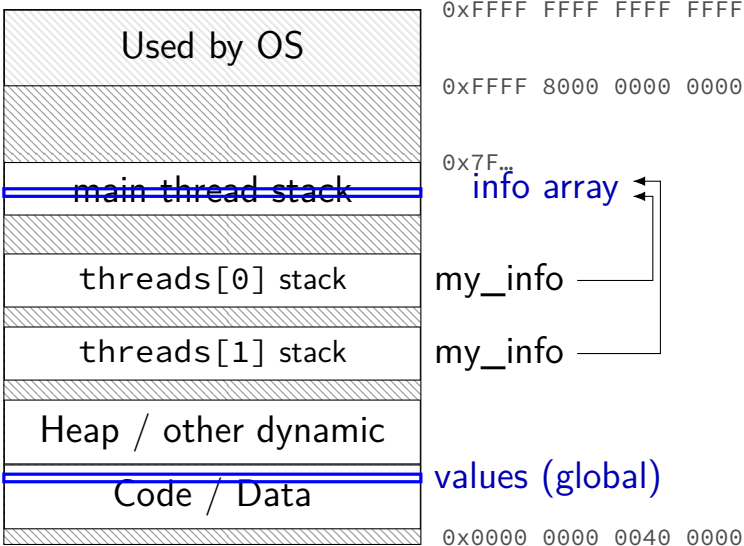
```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    struct ThreadInfo *my_info = (struct ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; ; }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; struct ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i) { pthread_join(threads[i], NULL); }
    return info[0].result + info[1].result;
}
```

my_info: pointer to sum_all's stack
only okay because sum_all waits!

# sum example (info struct)

```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    struct ThreadInfo *my_info =  (struct ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) { sum += values[i]; }
    my_info->result = sum;
    return NULL;
}
int sum_all() {
    pthread_t thread[2]; struct ThreadInfo info[2];
    for (int i = 0; i < 2; ++i) {
        info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, &info[i]);
    }
    for (int i = 0; i < 2; ++i) { pthread_join(threads[i], NULL); }
    return info[0].result + info[1].result;
}
```

# thread_sum memory layout (info struct)



# sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

# sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

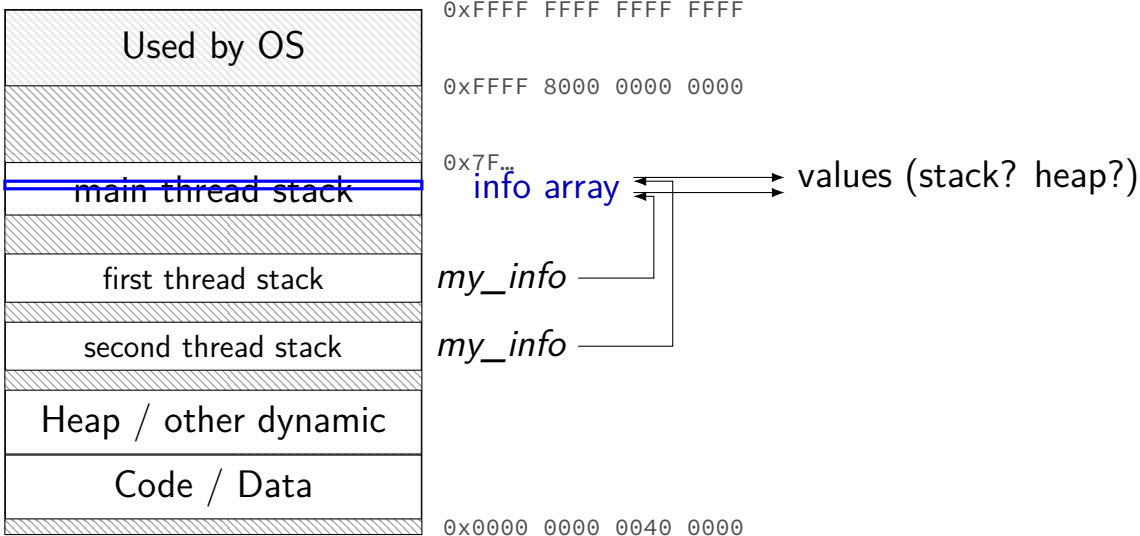
# sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

# sum example (to main stack)

```
struct ThreadInfo { int *values; int start; int end; int result };
void *sum_thread(void *argument) {
    ThreadInfo *my_info = (ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start; i < my_info->end; ++i) {
        sum += my_info->values[i];
    }
    my_info->result = sum;
    return NULL;

}
int sum_all(int *values) {
    ThreadInfo info[2]; pthread_t thread[2];
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&threads[i], NULL, sum_thread, (void *) &info[i]);
    }
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);
    return info[0].result + info[1].result;
}
```

# program memory (to main stack)



# sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result
void *sum_thread(void *argument) {
    ...
}

struct ThreadInfo *start_sum_all(int *values) {
    struct ThreadInfo *info = calloc(2, sizeof(struct ThreadInfo));
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    free(info);
    return result;
}
```

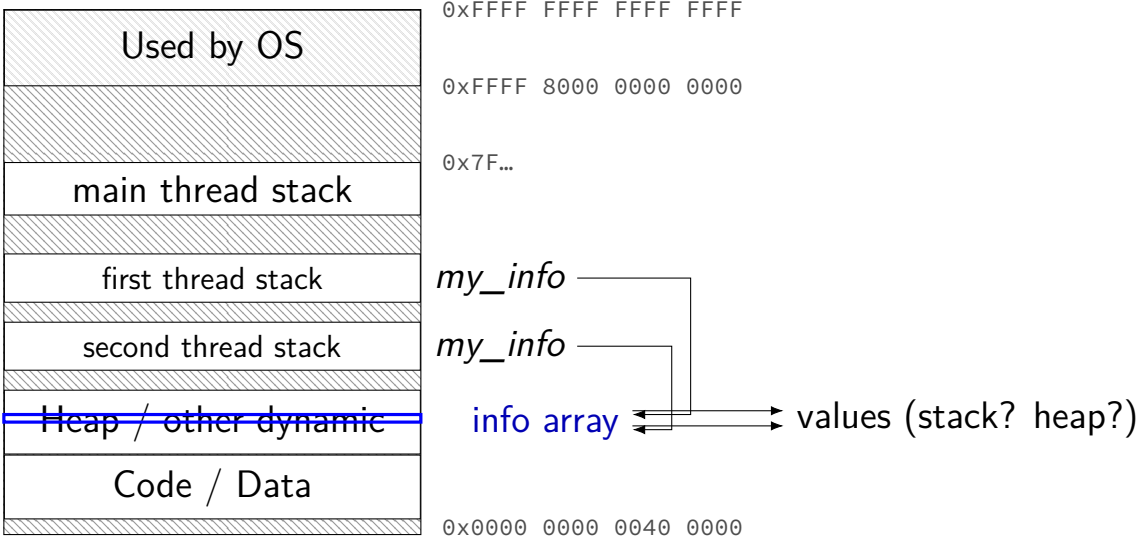
# sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result
void *sum_thread(void *argument) {
    ...
}

struct ThreadInfo *start_sum_all(int *values) {
    struct ThreadInfo *info = calloc(2, sizeof(struct ThreadInfo));
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    free(info);
    return result;
}
```

# sum example (on heap)

```
struct ThreadInfo { pthread_t thread; int *values; int start; int end; int result
void *sum_thread(void *argument) {
    ...
}

struct ThreadInfo *start_sum_all(int *values) {
    struct ThreadInfo *info = calloc(2, sizeof(struct ThreadInfo));
    for (int i = 0; i < 2; ++i) {
        info[i].values = values; info[i].start = i*512; info[i].end = (i+1)*512;
        pthread_create(&info[i].thread, NULL, sum_thread, (void *) &info[i]);
    }
    return info;
}

int finish_sum_all(ThreadInfo *info) {
    for (int i = 0; i < 2; ++i)
        pthread_join(info[i].thread, NULL);
    int result = info[0].result + info[1].result;
    free(info);
    return result;
}
```

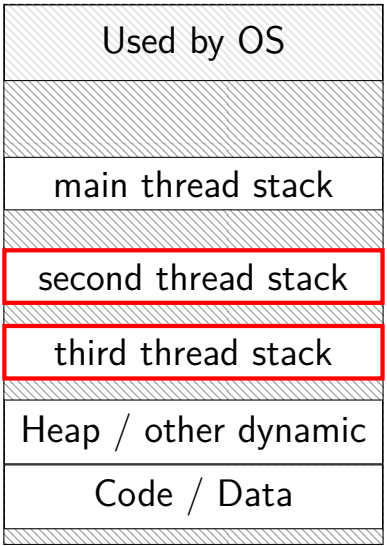
# thread_sum memory (heap version)



# what's wrong with this?

```
/* omitted: headers */
void *create_string(void *ignored_argument) {
  char string[1024];
  ComputeString(string);
  return string;
}
int main() {
  pthread_t the_thread;
  pthread_create(&the_thread, NULL, create_string, NULL);
  char *string_ptr;
  pthread_join(the_thread, (void**) &string_ptr);
  printf("string␣is␣%s\n", string_ptr);
}
```

# program memory




0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

dynamically allocated stacks

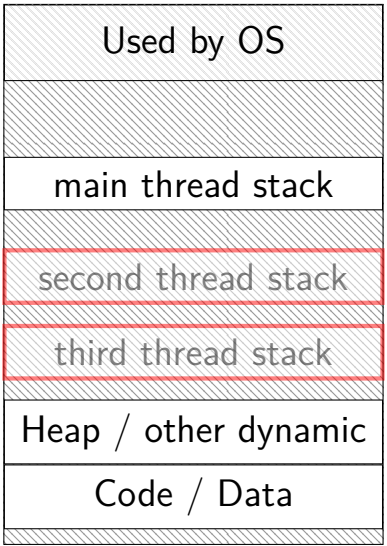
`char string[]` allocated here

`string_ptr` pointed to here

…stacks deallocated when threads exit/are joined

0x0000 0000 0040 0000

# program memory




0xFFFF FFFF FFFF FFFF

0xFFFF 8000 0000 0000

0x7F…

dynamically allocated stacks

`char string[]` allocated here

`string_ptr` pointed to here

…stacks deallocated when threads exit/are joined

0x0000 0000 0040 0000

# thread joining

pthread_join allows collecting thread return value

if you don't join joinable thread, then *memory leak*!

# thread joining

pthread_join allows collecting thread return value

if you don't join joinable thread, then *memory leak*!

avoiding memory leak?

always join...or

"detach" thread to make it not joinable

# pthread_detach

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_create(&show_progress_thread, NULL,
                   show_progress, NULL);

    /* instead of keeping pthread_t around to join thread later: */
    pthread_detach(show_progress_thread);
}

int main() {
    spawn_show_progress_thread();
    do_othe
    ...
}
```

detach = don't care about return value, etc.
system will deallocate when thread terminates

# starting threads detached

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setdetachstate(&attrs, PTHREAD_CREATE_DETACHED);
    pthread_create(&show_progress_thread, attrs,
                   show_progress, NULL);
    pthread_attr_destroy(&attrs);
}
```

# setting stack sizes

```
void *show_progress(void * ...) { ... }
void spawn_show_progress_thread() {
    pthread_t show_progress_thread;
    pthread_attr_t attrs;
    pthread_attr_init(&attrs);
    pthread_attr_setstacksize(&attrs, 32 * 1024 /* bytes */);
    pthread_create(&show_progress_thread, attrs,
                   show_progress, NULL);
}
```

# backup slides

# thread versus process state

thread state

- registers (including stack pointer, program counter)
- ...

process state

- address space
- open files
- process id
- list of thread states
- ...

# process info with threads

parent process info

| | |
|---|---|
| thread infos | `thread 0: {PC = 0x123456, rax = 42, rbx = …}`<br>`thread 1: {PC = 0x584390, rax = 32, rbx = …}`<br>… |
| page tables | |
| open files | `fd 0: …`<br>`fd 1: …` |
| … | … |

# Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers *can be shared*

- e.g. shared open files: open fd 4 in one task → all sharing can use fd 4

`fork()`-like system call "clone": *choose what to share*

- `clone(0, ...)` — similar to `fork()`
- `clone(CLONE_FILES, ...)` — like fork(), but **sharing** open files
- `clone(CLONE_VM, new_stack_pointer, ...)` — like fork(), but **sharing** address space

# Linux idea: task_struct

Linux model: single "task" structure = thread

pointers to address space, open file list, etc.

pointers *can be shared*

- e.g. shared open files: open fd 4 in one task → all sharing can use fd 4

`fork()`-like system call "clone": *choose what to share*

- `clone(0, ...)` — similar to `fork()`
- `clone(CLONE_FILES, ...)` — like fork(), but **sharing** open files
- `clone(CLONE_VM, new_stack_pointer, ...)` — like fork(), but **sharing** address space

advantage: no special logic for threads (mostly)

# aside: alternate threading models

we'll talk about *kernel threads*

OS scheduler deals **directly** with threads

alternate idea: library code handles threads

kernel doesn't know about threads w/in process

*hierarchy* of schedulers: one for processes, one within each process

not currently common model — awkward with multicore