Review

Quiz week 13, Q2

- Suppose a 5-stage pipelined processor, similar to the design discussed in lecture, executes a program with 100 million instructions. If there were no hazards, this program would execute in about 100 million cycles.
- Suppose that we do not have forwarding (also known as bypassing) from the memory stage to the execute stage. 30% of the program's instructions use the data cache, either loading a value of from data cache or storing a value. Of the those data cache using instructions, 70% are loads. 10% of those loads are followed by an instruction that has a data dependency on the load.
- Assuming no cache misses, we'd expect the program to execute around __% slower than the ideal 100 million cycle case.
- Answer: 2.1
  - 21% of instructions are loads, 10% of those will have a 1-cycle stall, so 2.1%
- 1 cycle stall because EX stage needs the load result of the instruction just in front of it, but that won't be ready for another cycle

ld
- `IF ID EX MEM WB`

add
- ~~`IF ID EX EX MEM WB`~~

IF ID ID EX

ld a(%r8), %r7

add q %r9, %r7

prod

ld IF ID EX MEM WB

r8 r7

IF ID EX MEM

r7

Quiz week 13, Q4

- Suppose we start with the 5-staged pipelined processor with forwarding and branch prediction discussed in lecture, but make some modifications:
- We want a larger L1 data cache, whose access time doesn't fit into one cycle. (Or else the entire pipeline would have to run at a slower clock speed). So we break the memory stage into two stages. So our pipeline consists of F-D-E-M1-M2-W. Assume the two part memory stages use the same inputs and outputs as an unsplit memory stage, and they need the inputs (the address to access and any value to store) near the beginning of the memory part 1 stage and only have outputs (any value loaded) near the end of the memory part 2 stage.
- Q4: Suppose the processor design change described above increases the data cache hit rate from 90 to 99% for the programs being run, and a cache miss costs 100 cycles. But this change also causes 10% of the instructions the processor runs to take an extra cycle from stalling (because of a data hazard involving the M2 stage) when previously no instructions required stalling. Which of the following best describes how much the decision help or hurt performance?
    - Original: 90% of instructions take 1 cycle; 10% take 100 cycles
        - So for N instructions, we need 1.0*N*1 + 0.1*N*100 = 1 + 10 = 11 N
    - New?

30% loads

$1.0N*1 + 0.3*0.1*N*100 = 1+3 = 4N$

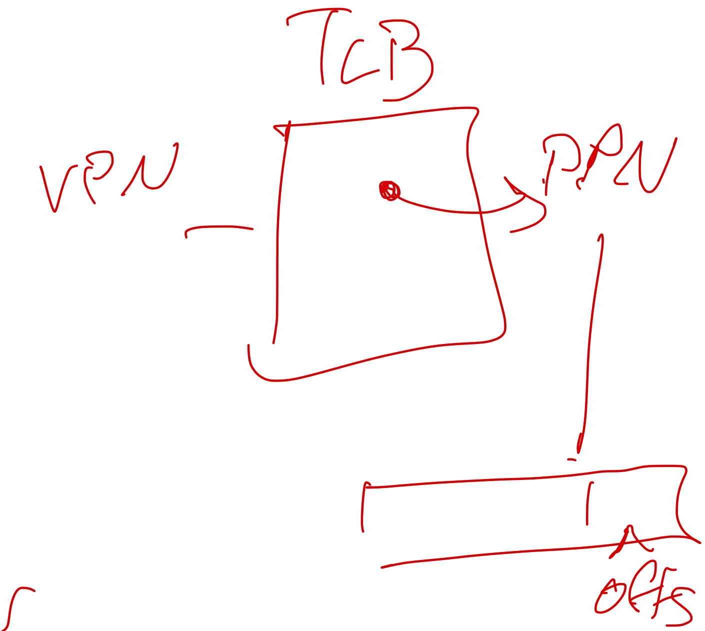
Quiz week 13, Q4

- Suppose we start with the 5-staged pipelined processor with forwarding and branch prediction discussed in lecture, but make some modifications:
- We want a larger L1 data cache, whose access time doesn't fit into one cycle. (Or else the entire pipeline would have to run at a slower clock speed). So we break the memory stage into two stages. So our pipeline consists of F-D-E-M1-M2-W. Assume the two part memory stages use the same inputs and outputs as an unsplit memory stage, and they need the inputs (the address to access and any value to store) near the beginning of the memory part 1 stage and only have outputs (any value loaded) near the end of the memory part 2 stage.
- Q4: Suppose the processor design change described above increases the data cache hit rate from 90 to 99% for the programs being run, and a cache miss costs 100 cycles. But this change also causes 10% of the instructions the processor runs to take an extra cycle from stalling (because of a data hazard involving the M2 stage) when previously no instructions required stalling. Which of the following best describes how much the decision help or hurt performance?
    - Original: 90% of instructions take 1 cycle; 10% take 100 cycles
        - So for N instructions, we need ~~1.0*N*1 + 0.1*N*100 = (1 + 10)N = 11N~~ 4N
    - New: 1.0*N*1 + 0.01*N*100 + 0.1*N*1 = (1+1+.1)N = 2.~~1N~~ 2

0.3

$(1 + 0.3 + 0.1)N$ $1.4N = N$

TLB/cache access



TLB : VPN in PPN out
indep of how many levels of PT

32-bit VAddr 20 bit Paddr 8 KB pages

128-entry TLB and FA → 0 index bits

16-way index bits $= \log \frac{128}{16}$ $\frac{2^7}{2^4} = 2^3$ 3 bits index 16 tag bits

TLB/cache access

20 bit PA

8 KB, 2-way cache

2^{13}

32B blocks

2^5

How many cache blocks?

$2^{13}/2^5 = 2^8$ blocks

256

2^1 way = 128 sets

2^7

PA

20 bits

T	I	O
8	7	5

Networking – key concepts

- Mailbox model with acks (different versions of protocol, eg what's in the slides vs. what we did in lab)
- How simple mailbox model needs to be modified to deal with disruptions in delivery (lost messages, reordered messages, repeated messages)
  delayed too
- What the different network layers do
- How a message to a server name (e.g., portal.cs.virginia.edu) finds the correct IP address
- How routers route packets from source to destination

Crypto key concepts

- Symmetric encryption
  - and MACs
- Asymmetric encryption
  - And signatures
  - And certificates and how certificate authorities prove validity of a party's public key (assuming you trust the certificate)
- How replay/protocol attacks work and how to prevent them
  - See also secure channels lab
- Diffie-Hellman and TLS

2 keys for encryption + MAC

1 public/private key pair per party

Send message → encrypt w/ recipient's public key
they decrypt w/ private key

Sign message → encrypting w/ private key
a hash

Recipient verifies w/ senders public key

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;
}
```

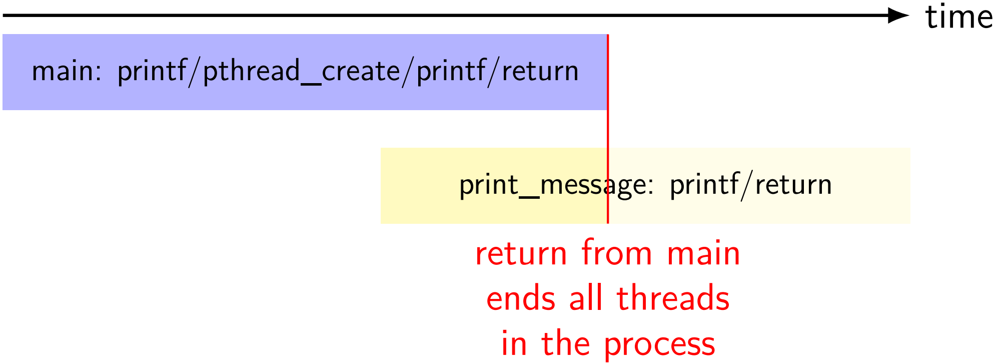
My machine: outputs `In the thread` *about 4% of the time*.

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;
}
```

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

sum example (only globals)

```c
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

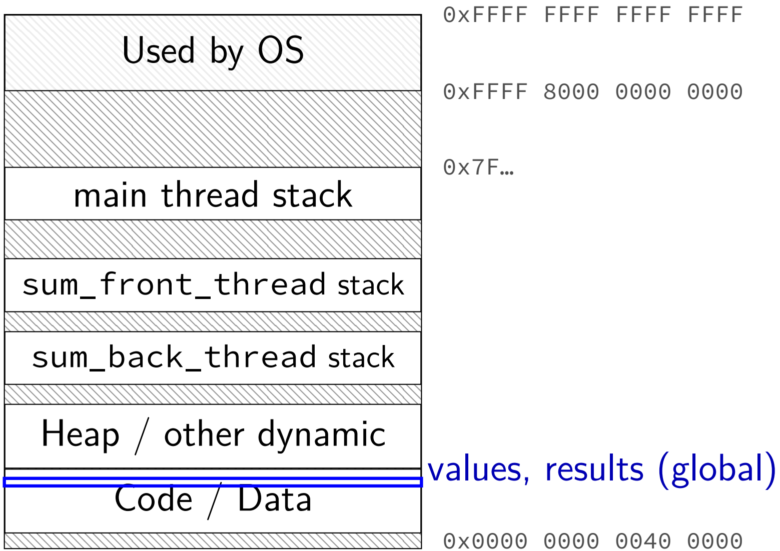
```
int values[1024];  in
void *sum_front(void 
    int sum = 0;
    for (int i = 0; i < 512; ++i) { sum += values[i]; }
    results[0] = sum;
    return NULL;
}
void *sum_back(void *ignored_argument) {
    int sum = 0;
    for (int i = 512; i < 1024; ++i) { sum += values[i]; }
    results[1] = sum;
    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 e

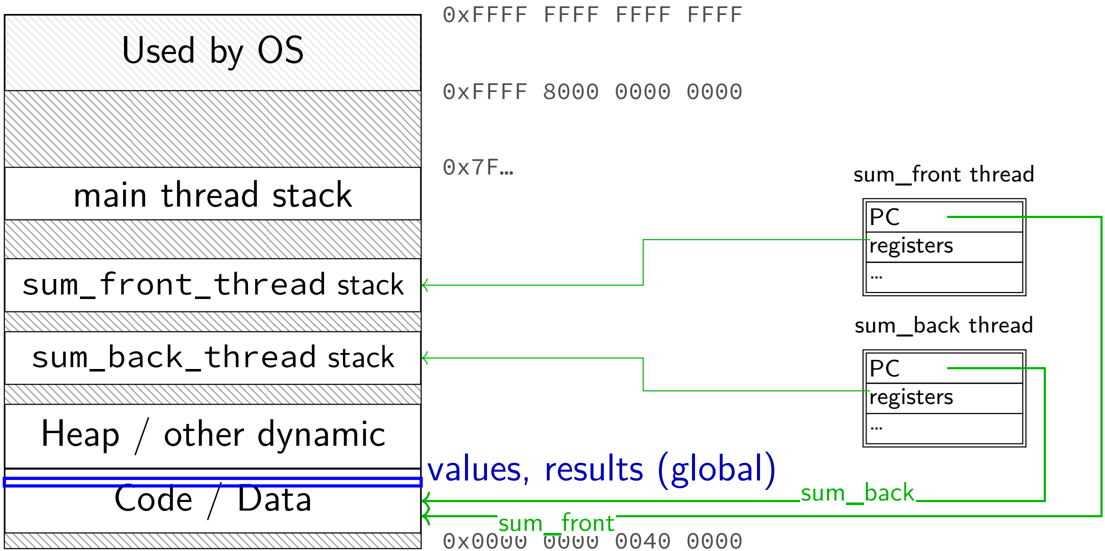
values returned from threads
via global array instead of return value
(partly to illustrate that memory is shared,
partly because this pattern works when we don't join (later))

```
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



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)

```
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];
}
```

values, results: global variables — 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; 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)

values: global variable — shared

```
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)

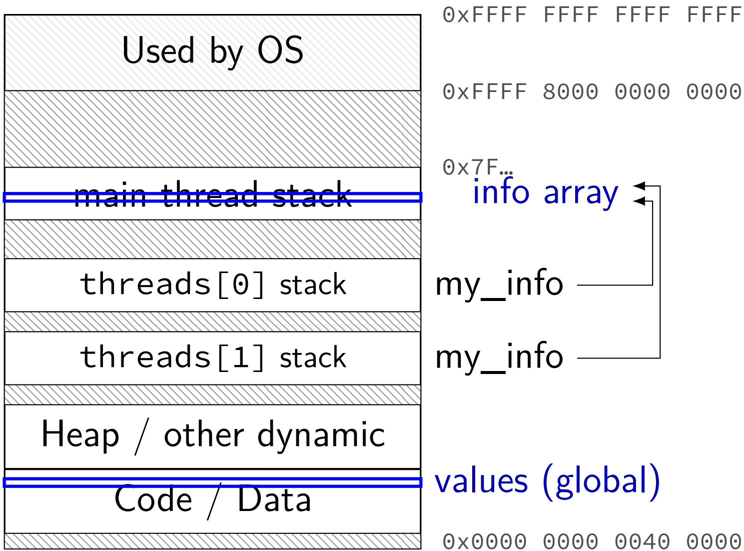
```
int values[1024];
struct ThreadInfo {
    int start, end, result;
};
void *sum_thread(void *argument) {
    struct ThreadInfo *my_info = (struct ThreadInfo *) argument;
    int sum = 0;
    for (int i = my_info->start;                              ; }
    my_info->result = sum;
    return NULL;
}
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;
}
```

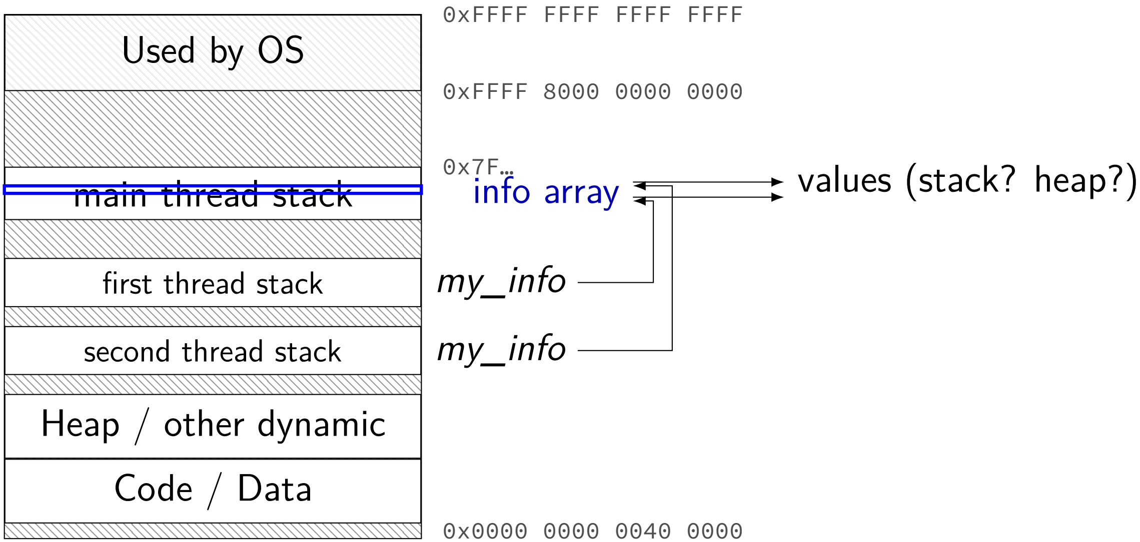
sum example (to main stack)

```
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;
}
```

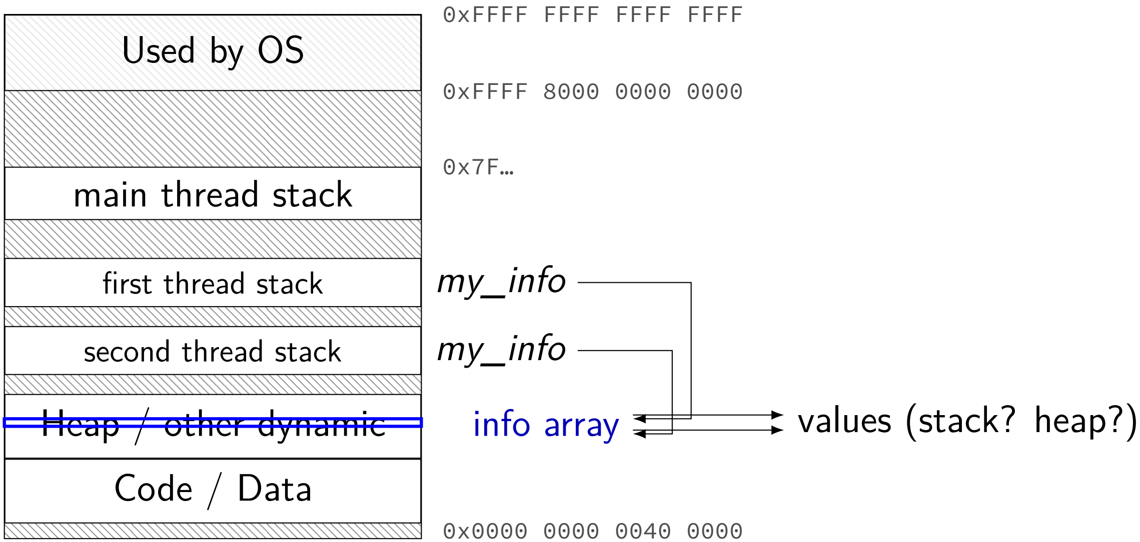
sum example (on heap)

```
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;
}
```

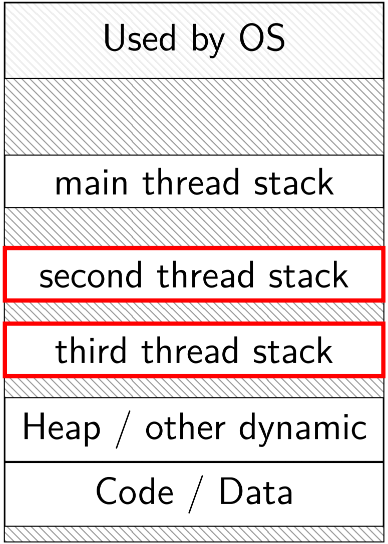
thread_sum memory (heap version)



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

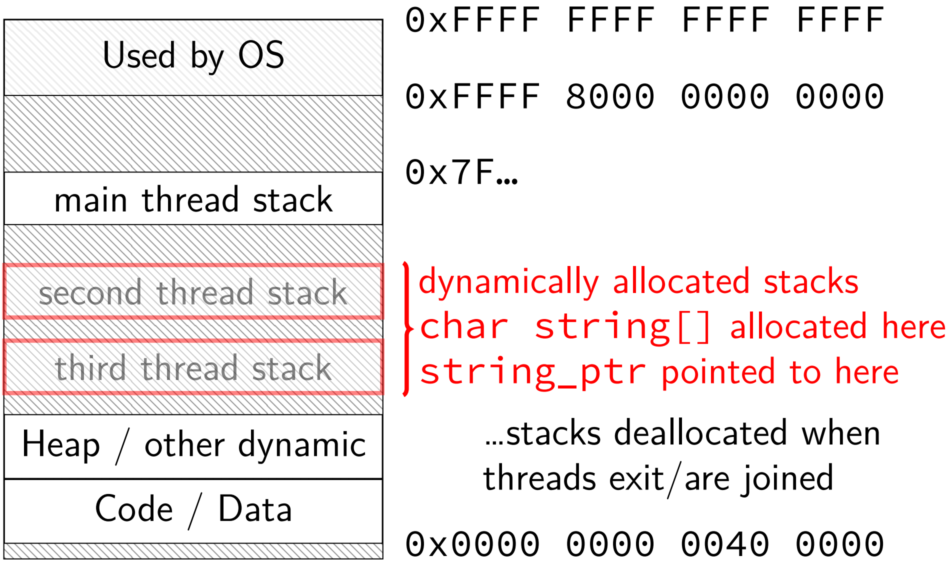
0x7F…

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



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

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}
Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}
Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

rule: never touch `buffer` without acquiring lock

otherwise: what if two threads simultaneously en/dequeue?

(both use same array/linked list entry?)

(both reallocate array?)

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}
Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

check if not empty

if so, dequeue

okay because have lock

other threads can**not** dequeue here

unbounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready;
UnboundedQueue buffer;

Produce(item) {
    pthread_mutex_lock(&lock);
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}
Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_mutex_unlock(&lock);
    return item;
}
```

wake one Consume thread
if any are waiting

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;
Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(&space_ready, &lock); }
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}
Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;
Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(&space_ready, &lock); }
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}
Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;
Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(&space_ready, &lock); }
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
```

correct (but slow?) to replace with:

`pthread_cond_broadcast(&space_ready);`

(just more "spurious wakeups")

```
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

bounded buffer producer/consumer

```
pthread_mutex_t lock;
pthread_cond_t data_ready; pthread_cond_t space_ready;
BoundedQueue buffer;
Produce(item) {
    pthread_mutex_lock(&lock);
    while (buffer.full()) { pthread_cond_wait(&space_ready, &lock); }
    buffer.enqueue(item);
    pthread_cond_signal(&data_ready);
    pthread_mutex_unlock(&lock);
}
Consume() {
    pthread_mutex_lock(&lock);
    while (buffer.empty()) {
        pthread_cond_wait(&data_ready, &lock);
    }
    item = buffer.dequeue();
    pthread_cond_signal(&space_ready);
    pthread_mutex_unlock(&lock);
    return item;
}
```

correct but slow to replace `data_ready` and `space_ready` with 'combined' condvar `ready` and use broadcast
(just more "spurious wakeups")

monitor pattern

```
pthread_mutex_lock(&lock);
while (!condition A) {
    pthread_cond_wait(&condvar_for_A, &lock);
}
... /* manipulate shared data, changing other conditions */
if (set condition A) {
    pthread_cond_broadcast(&condvar_for_A);
    /* or signal, if only one thread cares */
}
if (set condition B) {
    pthread_cond_broadcast(&condvar_for_B);
    /* or signal, if only one thread cares */
}
...
pthread_mutex_unlock(&lock)
```

monitors rules of thumb

never touch shared data without holding the lock

keep lock held for *entire operation*:

verifying condition (e.g. buffer not full) *up to and including* manipulating data (e.g. adding to buffer)

create condvar for every kind of scenario waited for

always write *loop* calling cond_wait to wait for condition X

broadcast/signal condition variable *every time you change X*

monitors rules of thumb

never touch shared data without holding the lock

keep lock held for *entire operation*:

- verifying condition (e.g. buffer not full) *up to and including*
- manipulating data (e.g. adding to buffer)

create condvar for every kind of scenario waited for

always write *loop* calling cond_wait to wait for condition X

broadcast/signal condition variable *every time you change X*

correct but slow to...

- broadcast when just signal would work
- broadcast or signal when nothing changed
- use one condvar for multiple conditions

reader/writer problem

some shared data

only one thread modifying (read+write) at a time

read-only access *from multiple threads* is safe

reader/writer problem

some shared data

only one thread modifying (read+write) at a time

read-only access *from multiple threads* is safe

could use lock — but doesn't allow multiple readers

reader/writer locks

abstraction: lock that distinguishes readers/writers

operations:

- read lock: wait until no writers
- read unlock: stop being registered as reader
- write lock: wait until no readers and no writers
- write unlock: stop being registered as writer

reader/writer locks

abstraction: lock that distinguishes readers/writers

operations:

- read lock: wait until no writers
- read unlock: stop being registered as reader
- write lock: wait until *no readers and no writers*
- write unlock: stop being registered as writer

pthread rwlocks

```
pthread_rwlock_t rwlock;
pthread_rwlock_init(&rwlock, NULL /* attributes */);
...
    pthread_rwlock_rdlock(&rwlock);
    ...  /* read shared data */
    pthread_rwlock_unlock(&rwlock);

    pthread_rwlock_wrlock(&rwlock);
    ...  /* read+write shared data */
    pthread_rwlock_unlock(&rwlock);

...
pthread_rwlock_destroy(&rwlock);
```

Before pthread rwlocks, you used to have to implement these protocols using condvars, ensuring one reader at a time; no writers concurrent with readers; the last reader out signaled a waiting writer; and when a writer departed, it broadcast to all the waiting readers