More on Multi-threading
5 steps of thread_create

• More on these 5 steps

• 4 steps of thread_yield

• Semaphores for testing

• Inter-process communication (IPC)
int main() {
    thread_init();
    thread_create(test_code, "thread 1", 16 * 1024);
    thread_create(test_code, "thread 2", 16 * 1024);
    test_code("main thread");
}

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
Question: Are other outputs possible?

```c
int main() {
    thread_init();
    thread_create(test_code, "thread 1", 16 * 1024);
    thread_create(test_code, "thread 2", 16 * 1024);
    test_code("main thread");
}

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
```

thread1 here: 0
thread2 here: 0
thread1 here: 1
main thread here: 0
thread2 here: 1
thread1 here: 2
main thread here: 1
thread2 here: 2
thread1 done
main thread here: 2
thread2 done
main thread done
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
}

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
int main() {
    thread_init();
    thread_create(test_code, "thread 1", 16 * 1024);
    thread_create(test_code, "thread 2", 16 * 1024);
    test_code("main thread");
}

ctx_start: // step 2/5: thread_create() calls ctx_start()
    ...
    // save registers on the stack with store instructions
    mv sp, a1
    call ctx_entry
int main() {
    thread_init();
    thread_create(test_code, "thread 1", 16 * 1024);
    thread_create(test_code, "thread 2", 16 * 1024);
    test_code("main thread");
}

ctx_start:        // step 3/5: thread_create() passes the new
...             // stack pointer as 2nd argument to ctx_start()
    mv sp, a1     // ctx_start() modifies sp to its 2nd argument
    call ctx_entry

stack
    main
    thread_create
    saved registers

old stack pointer -> new stack pointer

heap
    16KB of garbage data

data
code
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
}

ctx_start:
...
    mv sp, a1
    call ctx_entry  // step 4/5: call function ctx_entry()
int main() {
  thread_init();
  thread_create(test_code, "thread 1",
               16 * 1024);
  thread_create(test_code, "thread 2",
               16 * 1024);
  test_code("main thread");
}

void test_code(void *arg) {
  for (int i = 0; i < 3; i++) {
    printf("%s here: %d\n", arg, i);
    thread_yield();
  }
  printf("%s done\n", arg);
  thread_exit();
}
• 5 steps of `thread_create`

More on these 5 steps

• 4 steps of `thread_yield`

• **Semaphores** for testing

• Inter-process communication (IPC)
Question: Is Malloc a system call?

Step 1/5
thread_create allocates 16KB of memory on heap

16KB of garbage data
Before any allocation: Heap is **empty**

- **Stack**
  - main
  - thread_create

- **Data**
- **Code**

**Break pointer**

❌ access this address will cause memory exception
System call \texttt{sbrk()}  

// by invoking this system call  
void* old = sbrk(64 * 1024);
malloc() is a C library function

// This is compiled with a malloc()
// implementation in the C compiler.
malloc(16 * 1024);
// when malloc() is first called
malloc(16 * 1024);

// malloc() first gets a larger
// region using sbrk, say 64KB
void* old = sbrk(64 * 1024);
// when malloc() is first called
malloc(16 * 1024);

// malloc() first gets a larger
// region using sbrk, say 64KB
void* old = sbrk(64 * 1024);

// then malloc creates data structures
// and assigns 16KB to the application
Second `malloc()`, step 1/1

// when `malloc()` is first called
`malloc(16 * 1024);`

// This time, `malloc()` may not
// not need to call `sbrk` again.
`malloc(8 * 1024);`
Question: Is **Malloc** a system call?

The GNU C Library

malloc maintains the data structures in the heap.

sbrk moves break pointer.

new break pointer

old break pointer

stack

main

thread_create

heap

data

code
Question: is it necessary to save all registers?

Step 2/5
**ctx_start** saves registers on the stack
How to save the old stack pointer?

```c
struct thread {
    void* sp;
    ...
}

// say we have the pointers to
// the current and next threads
struct thread* current, next;

ctx_start(&current->sp, next->sp);
// In ctx_start:
//    sw sp, 0(a0)    /* sp -> 0(a0) */
//    mv sp, a1       /* a1 -> sp */
```
int main() {
    thread_init();
    thread_create(test_code, "thread 1", 16 * 1024);
    thread_create(test_code, "thread 2", 16 * 1024);
    test_code("main thread");
}

ctx_start:
    ...
    mv sp, a1
    call ctx_entry
    call test_code
    call thread_exit // it is possible to replace call ctx_entry
    // with these two instructions
C programs do something similar

// Every C program has a main() function.
// The C program is compiled with some assembly by the compiler.

li sp, {some address}  // initialize stack pointer
call main              // return value of main is in register a0
call exit              // a0 is also the first parameter to exit

// exit() is a C library function which invokes a system call
// In RISC-V, system calls are invoked by an ecall instruction
// You will meet the ecall instruction in P3.
• 5 steps of `thread_create`
• More on these 5 steps

• 4 steps of `thread_yield`

• **Semaphores** for testing

• Inter-process communication (IPC)
int main() {
    thread_init();
    thread_create(test_code, "thread 1",
                  16 * 1024);
    thread_create(test_code, "thread 2",
                  16 * 1024);
    test_code("main thread");
}

void test_code(void *arg) {
    for (int i = 0; i < 3; i++) {
        printf("%s here: %d\n", arg, i);
        thread_yield();
    }
    printf("%s done\n", arg);
    thread_exit();
}
void test_code(void *arg) {
    ...
    thread_yield();
    ...
}

ctx_switch:

    ... // save registers on the stack with store instructions
    sw sp, 0(a0)
    mv sp, a1
    ...
    ret
void test_code(void *arg) {
    ...
    thread_yield();
    ...
}

ctx_switch:       // ctx_switch(&current->sp, next->sp)
    ...
    sw sp, 0(a0)  // save old stack pointer
    mv sp, a1     // switch to the new stack pointer
    ...
    ret
void test_code(void *arg) {
    ...
    thread_yield();
    ...
}

ctx_switch:
    ...
    sw sp, 0(a0)
    mv sp, a1
    // restore the registers saved in the main thread
    ret
int main() {
    thread_init();
    thread_create(test_code, "thread 1", 16 * 1024);
    thread_create(test_code, "thread 2", 16 * 1024);
    test_code("main thread");
}

ctx_switch:
   ...
   → ret  // step 4/4: return to thread_create()
thread_create:
   ...
   → ret  // step 4/4: return to main()
- 5 steps of `thread_create`
- More on these 5 steps
- 4 steps of `thread_yield`

Semaphores for testing

- Inter-process communication (IPC)
Semaphores as counters

// Allocate a counter
void sema_init(struct sema *sema, unsigned int count);

// Increment the counter by 1
void sema_inc(struct sema *sema);

// Wait until counter > 0, then decrement counter by 1
void sema_dec(struct sema *sema);

// Release the counter
bool sema_release(struct sema *sema);
Bounded buffer **producer-consumer**

A **fixed length array** as a buffer holding items between producer and consumer threads

<table>
<thead>
<tr>
<th>Index</th>
<th>#1</th>
<th>#2</th>
<th>#3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bounded buffer producer-consumer

Now the buffer is full and all producer threads need to wait.
Bounded buffer producer-consumer

Say #5 consumes item#1 and #2; #7 consumes item #3; Now the buffer is empty and consumer threads need to wait.
Implement a producer-consumer with 3 semaphores.

```c
#define NBLOTS 3
static struct sema_s_empty, s_full, s_lock;
static unsigned int ln, out;
static char *slots[NBLOTS];

static void producer(void *arg)
{
    int i = 0;
    // First make sure there’s an empty slot.
    sema_dec(&s_empty);
    // Now add an entry to the queue
    sema_lock(&s_lock);
    slots[i] = arg;
    if (ln == NBLOTS) in = 0;
    sema_inc(&s_lock);
    // Finally, signal consumers
    sema_inc(&s_full);
}

static void consumer(void *arg)
{
    unsigned int i;
    for (i = 0; i < 3; i++)
    // First make sure there’s something in the buffer
    sema_dec(&s_full);
    // Now grab an entry from the queue
    sema_lock(&s_lock);
    void *x = slots[out++];
    printf("Got \"%s\". Arg: \n");
    if (out == NBLOTS) out = 0;
    sema_inc(&s_lock);
    // Finally, signal producers
    sema_inc(&s_empty);
}

int main(int argc, char **argv)
{
    thread_init();
    sema_init(&s_lock, 1);
    sema_init(&s_full, 0);
    sema_init(&s_empty, NBLOTS);

    thread_create(consumer, "consumer 1", 16 * 1024);  
    producer("producer 1");
    // Code should never reach here since producer is an infinite loop
    thread_exit();
    return 0;
}
```
Hints for `struct sema`

- Each semaphore maintains a queue of waiting threads.
  - consider the waiting producer/consumer just mentioned
- In P1, you need to implement more tests, such as
  - multi-reader/writer lock
  - dining philosophers
  - etc.
• 5 steps of `thread_create`
• More on these 5 steps
• 4 steps of `thread_yield`
• **Semaphores** for testing

⇒ Inter-process communication (IPC)
UNIX System V IPC

System V IPC is the name given to three interprocess communication mechanisms that are widely available on UNIX systems: message queues, semaphore, and shared memory.

https://man7.org/linux/man-pages/man7/svipc.7.html
Message queue example

UI thread in zoom

Microphone thread in zoom

User-level

Kernel-level

Operating Systems Kernel
Message queue example

UI thread in zoom

Microphone thread in zoom

User-level

Kernel-level

Operating Systems Kernel

System call:
I want a message queue!
Message queue example

UI thread in zoom

message queue

Microphone thread in zoom

User-level

Kernel-level

Alright!

System call:
I want a message queue!

Operating Systems Kernel
Message queue example

UI thread in zoom

message queue

Microphone thread in zoom

User-level

Kernel-level

System call:
Please send a “mute” message to the Microphone thread

Operating Systems Kernel
Message queue example

UI thread in zoom

User-level

Kernel-level

Microphone thread in zoom

Message queue

System call:
Please send a “mute” message to the Microphone thread

Alright!

Operating Systems Kernel
Message queue example

UI thread in Zoom

User-level

Kernel-level

System call:
Please send a "mute" message to the Microphone thread

message queue

Operating Systems Kernel

 Alright!

Microphone thread in Zoom

OK, I'll stop recording
Message queues in EGOS

- Earth layer
  - disk, screen, keyboard, interrupts, memory protection
- Grass layer
  - scheduler, system call and inter-process communication (IPC)
- Application layer
  - file system and shell (communicate through message queues)
  - shell commands: ls, mkdir, echo, cat, …
4411 projects design

- Earth layer
  - disk, screen, keyboard, interrupts, memory protection

- Grass layer
  - scheduler, system call and inter-process communication

- Application layer
  - file system and shell
  - shell commands: ls, mkdir, echo, cat, ...

Read yourself:
- Screen/keyboard: ~80 LOC
- Shell: ~50 LOC
- Other apps: ~150 LOC
High-level roadmap

- [ basic RISC-V CPU ] non-preemptive multi-threading
- [ + timer interrupt ] preemptive scheduling
- [ + privilege levels ] protection and isolation for processes
- [ + I/O bus controllers ] disk driver and file systems
Homework

• P1 is due on Sep 28.
  • multi-threading
  • semaphore
  • testing suite using semaphores