

CS 344: OPERATING SYSTEMS I

01.23: THREADS

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu



Oregon State
University

SAIL
Secure AI Systems Lab

NOTICE

- Deadlines
 - (1/23 11:59 PM) Programming assignment 1
 - (1/30 11:59 PM) Midterm quiz 1
 - (2/06 11:59 PM) Programming assignment 2

RECAP

- Part I: Process
 - Provide abstraction
 - What is a program?
 - What is a process?
 - How does OS run a program?
 - Offer standard libraries
 - How do we run (or stop) a process?
 - How does OS manage the process(es) we ran?
 - Manage resources
 - (Note) We will talk about this in the “scheduling” class

PRACTICE QUESTION: C

- **Static variables, functions, etc.**

- What will be the “google” stock price?
- What will be the prices of both stocks?
- What will be printed out to Terminal?
- What will be the prices of both stocks?
- What will be printed out to Terminal?

In C:

1. Static variables are accessible within a file
2. Increment / decrement operators
 - Prefix: increase the value first and exec
 - Postfix: exec and increase the value

Required headers...

```
static int google_stock = 2000;
```

```
int increase_price(int stock_price, int amount) {  
    stock_price = stock_price + amount;  
    google_stock = google_stock + amount;  
}
```

```
int main(void) {
```

Run --> `int apple_stock = 99;`

```
printf("Google stock price is %d\n", google_stock);
```

Run --> `printf("Apple stock price is %d\n", apple_stock++);`

```
apple_stock = increase_price(apple_stock, 50);
```

```
printf("Google stock price is %d\n", google_stock);
```

Run --> `printf("Apple stock price is %d\n", ++apple_stock);`

```
return 0;
```

```
}
```

PRACTICE QUESTION: C

- **Pointers and strings**

- What will be the value of “ilen”?
- What will be the value of “slen”?
- How many bytes “str” uses in memory?
- What will be the execution result?

In C:

1. Pointer variable store the address (size of the var will be 4-/8-byte)
2. Pointer is required to init. String with “=”
3. strlen() returns the number of chars
4. The actual string in C ends with “\0”

Required headers...

```
int main(void) {
```

```
    int slen = 0;
```

```
    int *iptr = NULL;
```

```
    Run --> char str = "Hello world!";
```

```
    Run -->     ilen = sizeof(iptr);
```

```
    Run -->     slen = strlen(str);
```

```
    printf("The length of this string is %d\n", slen);
```

```
    return 0;
```

```
}
```

PRACTICE QUESTION: C

• Pointer operations

- What will be printed out to Terminal?
- What will be printed out to Terminal?
- What will be printed out to Terminal?
- What will be printed out to Terminal?

In C:

1. Increasing pointer accesses the next address
!= Increasing pointer value (*ptr)+1
2. Pointer holds the address of a variable

Required headers...

```
int swap(int num1, int *num2) {  
    int temp = num1;  
    num1 = num2;  
    *num2 = temp;  
    return num1;  
}
```

```
int main(void) {  
    int val1 = 1;  
    int val2 = 2;  
    int vals[] = { 10, 20, 30, 40, 50 };  
    int *ptr = vals;
```

Run --> printf("Val1 / 2 / 3: %d, %d, %d\n", val1, val2, vals[0]);

Run --> printf("Val1 / 2: %d, %d\n", *(ptr+2), (*ptr)+2);

```
val1 = swap(val1, ptr);
```

Run --> printf("Val1 / 2 / 3: %d, %d, %d\n", val1, val2, vals[0]);

Run --> printf("Val1 / 2: %d, %d\n", *(ptr+2), (*ptr)+2);

```
return 0;
```

```
}
```

PRACTICE QUESTION: PROCESS

- **Segments (components) of a process**

- Which segment “counter1” is?
- Which segment “ret” is?
- Which segment “counter2” is?
- Which segment “buf” is?
- What are the counter1 and 2 values?
- Which segment “ret” is?
- Which segment “buf” is?

In Heap:

Memory fragmentation can happen
Memory leak can happen

Required headers...

```
#define BUFSIZE 512
```

```
static int counter1 = 0;
```

```
int my_function() {
```

```
Run --> int counter2 = 2;
```

```
Run --> char *buf = (char *) malloc(BUFSIZE * sizeof(char));
```

```
counter1 = counter1 + 1;
```

```
counter2 = counter2 - 1;
```

```
Run --> return counter2;
```

```
}
```

```
int main(void) {
```

```
Run --> int ret = 0;
```

```
Run --> ret = my_function();  
printf("Ret: %d\n", ret);
```

```
return 0;
```

```
}
```

TOPICS FOR TODAY

- Part I: Threads
 - Provide abstraction
 - What is a thread?
 - How is it different from a process?
 - How does OS run threads?
 - Offer standard libraries
 - How do we create/run/kill a thread?
 - How does OS manage the thread(s) we ran?
 - Manage resources
 - (Note) We will talk about this in the “scheduling” and “synchronization” classes

RUNNING MULTIPLE PROCESSES: WEB-SERVER EXAMPLE

- Amazon.com:
 - What does the webserver do?

WEB-SERVER EXAMPLE

- Amazon.com:
 - A user requests the website
 - A server accepts the connection
 - A server sends the webpage to the user
 - A user clicks something
 - A server sends the webpage as a response
 - ... (continue)

Pseudo code (server)

```
int main(void) {  
    // 1. server accepts the connection  
    connection = accepts(user-request, ...)  
  
    // 2. server sends the webpage to user  
    sends_webpage(connection, html-page)  
  
    // 3. server starts accepting the user requests  
    while (action = receive_request(connection)) {  
        if (action == login) {  
            if (!correct_credential(action.id, action.pw))  
                return -1; // return error, login fail  
            connection.login_success = 1;  
        }  
  
        ....  
    }  
  
    return 0; // halt the webserver, never reached  
}
```

WEB-SERVER EXAMPLE – CONT'D

- Amazon.com:
 - A user requests the website
 - A server accepts the connection
 - A server sends the webpage to the user
 - A user clicks something
 - A server sends the webpage as a response
 - ... (continue)

What would be a potential problem?

Pseudo code (server)

```
int main(void) {  
    // 1. server accepts the connection  
    connection = accepts(user-request, ...)  
  
    // 2. server sends the webpage to user  
    sends_webpage(connection, html-page)  
  
    // 3. server starts accepting the user requests  
    while (action = receive_request(connection)) {  
        if (action == login) {  
            if (!correct_credential(action.id, action.pw))  
                return -1; // return error, login fail  
            connection.login_success = 1;  
        }  
  
        ....  
    }  
  
    return 0; // halt the webserver, never reached  
}
```

WEB-SERVER EXAMPLE – CONT'D

- Amazon.com:

- A user requests the website
- A server accepts the connection
- A server sends the webpage to the user
- A user clicks something
- A server sends the webpage as a response
- ... (continue)

→ This procedure will be the **same** for all users
> **Multi-process** web-server

Pseudo code (server)

```
int main(void) {  
    // 1. server accepts the connection  
    connection = accepts(user-request, ...)  
  
    // 2. server sends the webpage to user  
    sends_webpage(connection, html-page)  
  
    // 3. server starts accepting the user requests  
    while (action = receive_request(connection)) {  
        if (action == login) {  
            if (!correct_credential(action.id, action.pw))  
                return -1; // return error, login fail  
            connection.login_success = 1;  
        }  
  
        ....  
    }  
  
    return 0; // halt the webserver, never reached  
}
```

MULTI-PROCESS WEB-SERVER EXAMPLE

- Amazon.com:

- A user requests the website
- A server accepts the connection
- A server sends the webpage to the user
- A user clicks something
- A server sends the webpage as a response
- ... (continue)

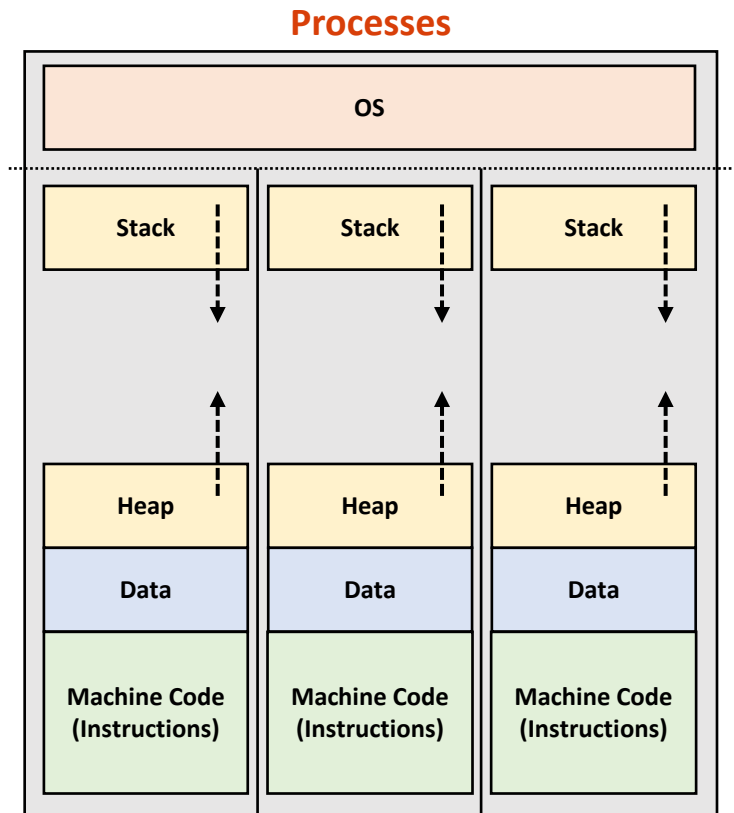
This procedure will be the **same** for all users
> **Multi-process** web-server

Pseudo code (server)

```
int main(void) {  
    while(connection = accepts(user-request, ...)) {  
        // fork: create a new process  
        switch(pid = fork()) {  
            case 0:  
                // server sends the webpage to user  
                sends_webpage(connection, html-page)  
  
                // server starts accepting the user requests  
                while (action = receive_request(connection)) {  
                    if (action == login) {  
                        if (!correct_credential(action.id, action.pw))  
                            return -1; // return error, login fail  
                        connection.login_success = 1;  
                    }  
                    ....  
                }  
            }  
        } // end of switch ...  
    } ...  
}
```

MULTI-PROCESS WEB-SERVER EXAMPLE: OS VIEW

- Amazon.com:
 - A user requests the website
 - A server accepts the connection
 - A server sends the webpage to the user
 - A user clicks something
 - A server sends the webpage as a response
 - ... (continue)

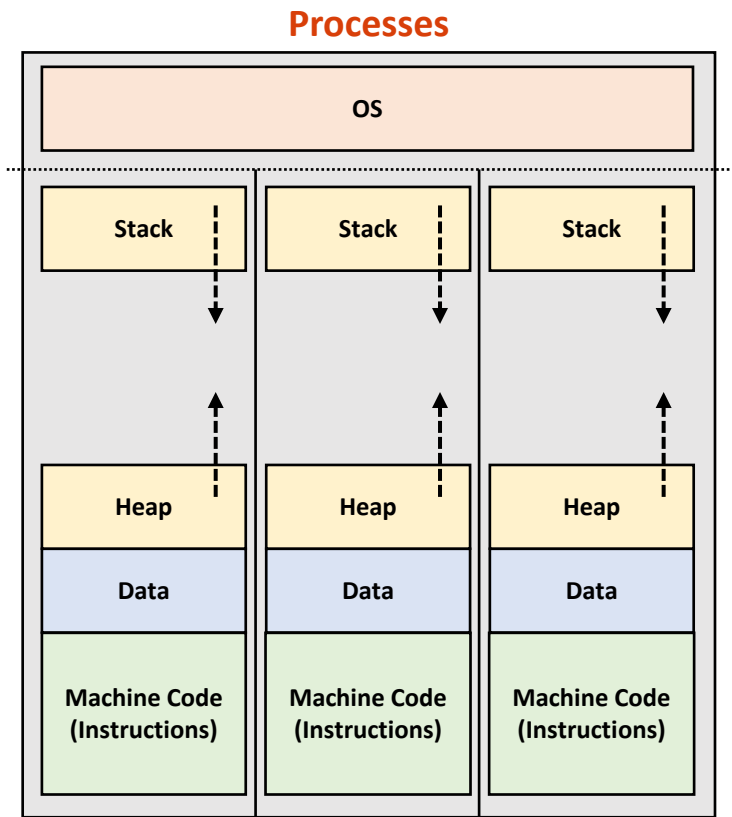


MULTI-PROCESS WEB-SERVER EXAMPLE: POTENTIAL ISSUES

- Data is not shared between processes
 - A user requests the website
 - ... (continue)

```
int main(void) {  
  
    // initialize some data in this block  
  
    while(connection = accepts(user-request, ...)) {  
        // fork: create a new process  
        switch(pid = fork()) {  
            case 0:  
                // server sends the webpage to user  
                sends_webpage(connection, html-page)  
  
                ...  
        }  
    }  
}
```

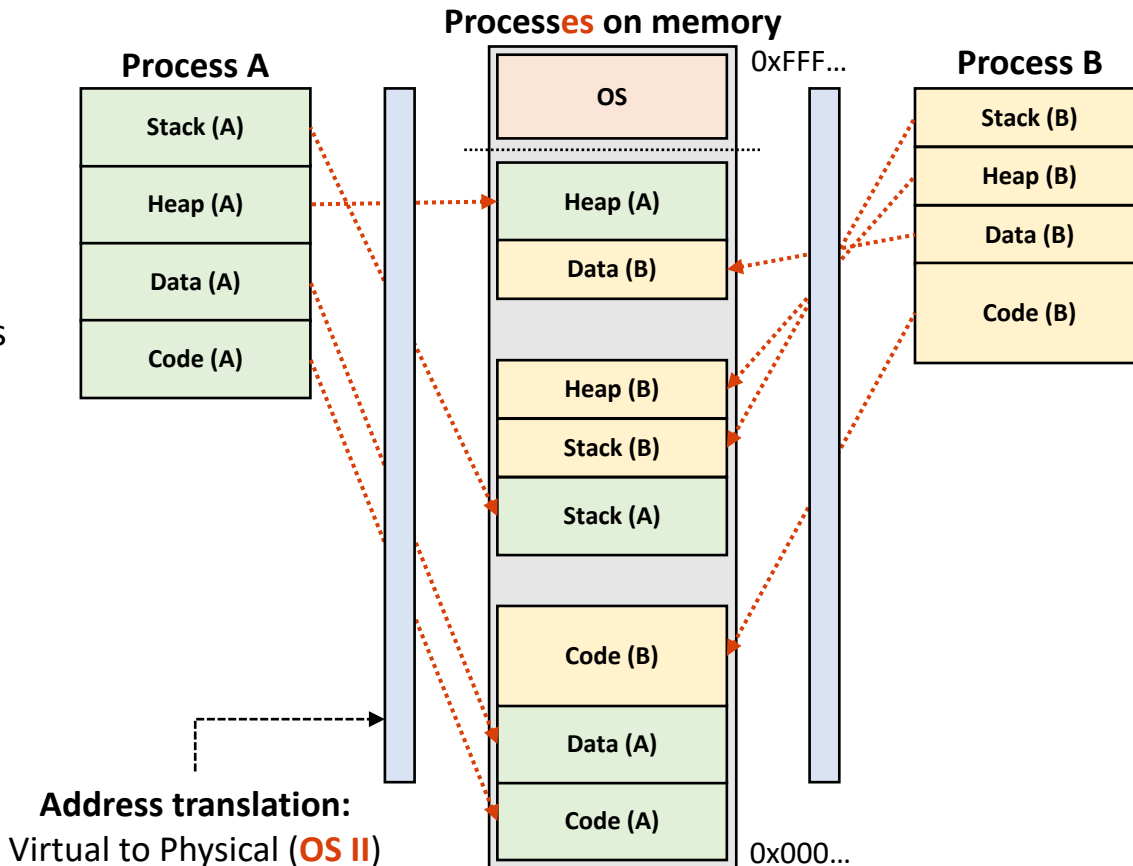
The data in the above block **won't be shared** between processes; each process will have a copy of the same data (*causes memory overhead)



NOTE: WHY ISN'T THE DATA SHARED BETWEEN PROCESSES?

• Process isolation

- No segment is shared
- Security reasons
 - Data breach
 - System crashes
 - Control other processes
 - ...
- Access: seg-faults!



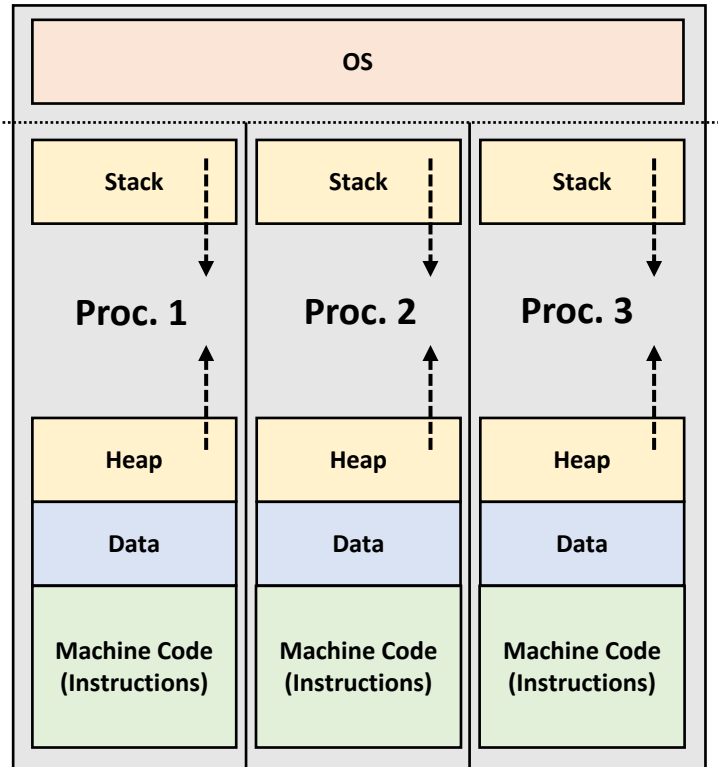
SOLUTION: THREADS

PROVIDE ABSTRACTION: A THREAD

- Thread
 - **Definition:** a smallest schedulable execution context
 - **Terminology:**
 - Smallest: it's much light-weight than a process
 - Schedulable execution context: one thread can run on a CPU at a time

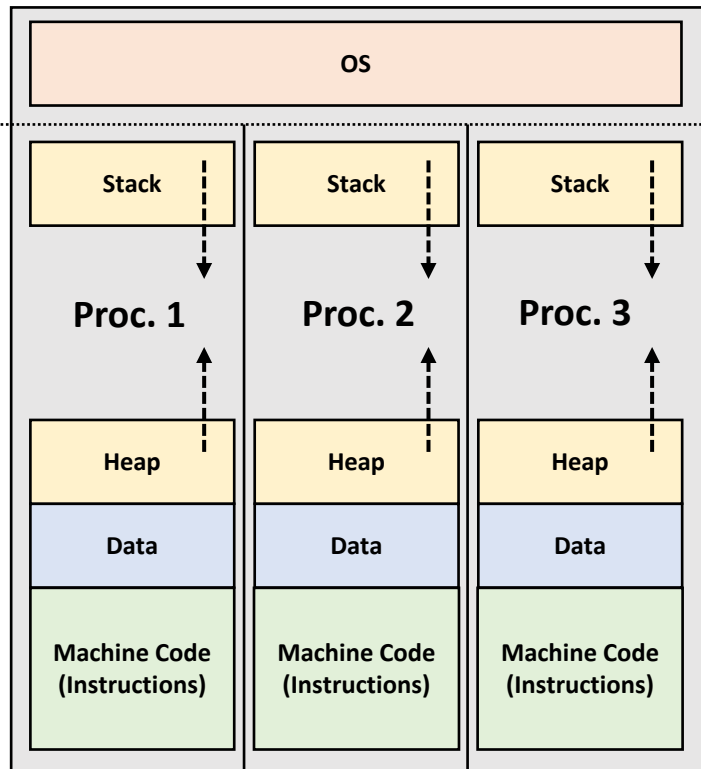
PROVIDE ABSTRACTION: A THREAD – CONT'D

Processes on memory

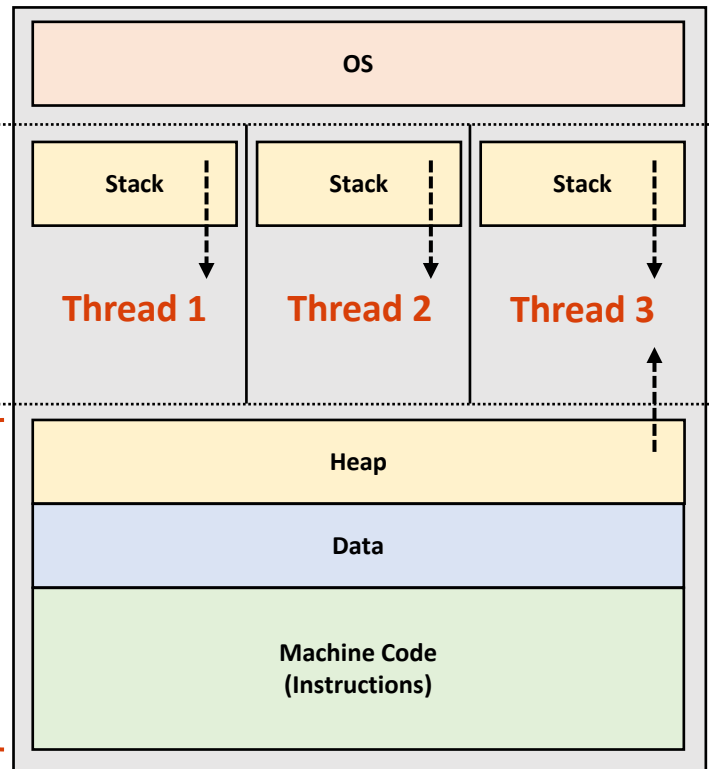


PROVIDE ABSTRACTION: A THREAD - CONT'D

Processes on memory



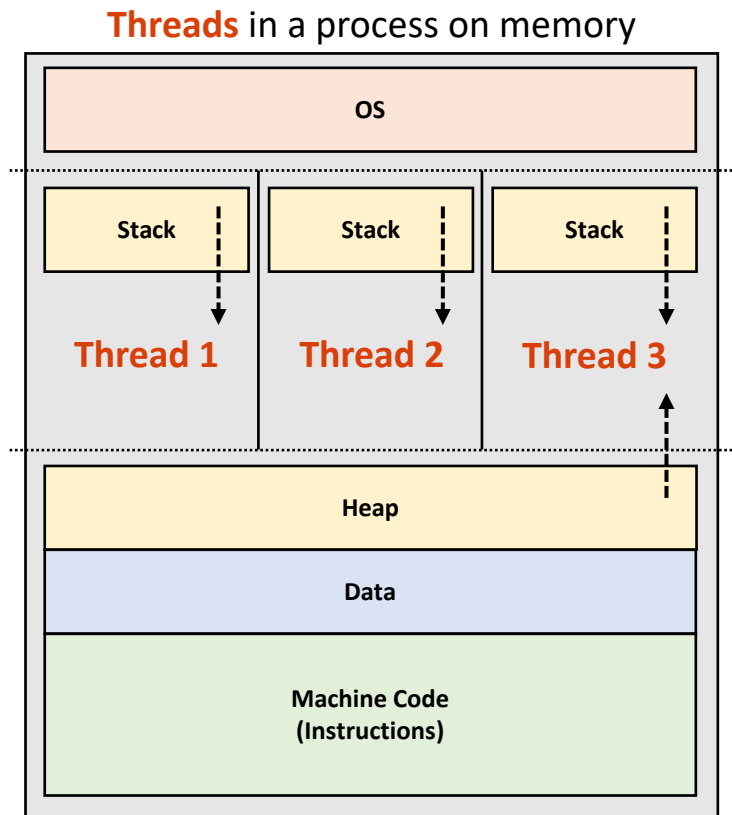
Threads in a process on memory



Reduce Duplications

PROVIDE ABSTRACTION: HOW IS IT DIFFERENT FROM A PROCESS?

- Threads share:
 - **Code** and **data** segments
 - **Heap** memory (ex. global variables)
 - Open files (ex. I/O access points)
- Threads **do not** share:
 - **Stack** segments, e.g.:
 - arguments passed when we launch them
 - local variables we initialize within them
 - return address, when they terminate ([OS II](#))
 - Running contexts, e.g.:
 - process state
 - stack pointer
 - ...



PROVIDE ABSTRACTION: HOW OS DEFINES A THREAD?

- (Linux) has “thread control block”

- Code

- Program counter
 - Instruction pointer

- Stack and heap

- Stack pointer
 - Heap pointer

- Running context

- Process state (ID, ...)
 - Execution flags
 - CPU # to run
 - (OS II) Scheduling policy
 - (OS II) Mem. virtualization

Thread Control Block: A set of information that OS requires to run a thread on a CPU, different from CPU vendors (ex. In Linux, it's the same: `task_struct`, [Link](#))

```
728 struct task_struct {
729 #ifdef CONFIG_THREAD_INFO_IN_TASK
730     /*
731      * For reasons of header soup (see current_thread_info()), this
732      * must be the first element of task_struct.
733      */
734     struct thread_info      thread_info;
735 #endif
736     unsigned int            __state;
737
738 #ifdef CONFIG_PREEMPT_RT
739     /* saved state for "spinlock sleepers" */
740     unsigned int            saved_state;
741 #endif
742
743     /*
744      * This begins the randomizable portion of task_struct. Only
745      * scheduling-critical items should be added above here.
746      */
747     randomized_struct_fields_start
748
749     void                    *stack;
750     refcount_t              usage;
751     /* Per task flags (PF_*), defined further below: */
752     unsigned int            flags;
753     unsigned int            ptrace;
754
755     struct sched_info       sched_info;
756
757     struct list_head        tasks;
758 #ifdef CONFIG_SMP
759     struct plist_node       pushable_tasks;
760     struct rb_node          pushable_dl_tasks;
761 #endif
762
763     struct mm_struct        *mm;
764     struct mm_struct        *active_mm;
765
766     /* Per-thread vma caching: */
767     struct vmacache        vmacache;
768
769 #ifdef SPLIT_RSS_COUNTING
770     struct task_rss_stat    rss_stat;
771 #endif
772
773     int                     exit_state;
774     int                     exit_code;
775     int                     exit_signal;
776     /* The signal sent when the parent dies: */
777     int                     pdeath_signal;
778     /* JOBCTL_*, siglock protected: */
779     unsigned long           jobctl;
780 }
```

A process and a thread are the same for OS

TOPICS FOR TODAY

- Part I: Threads
 - Provide abstraction
 - What is a thread?
 - How is it different from a process?
 - How does OS run threads?
 - Offer standard libraries
 - How do we create/run/kill a thread?
 - How does OS manage the thread(s) we ran?
 - Manage resources
 - (Note) We will talk about this in the “scheduling” and “synchronization” classes

OFFER STANDARD INTERFACE

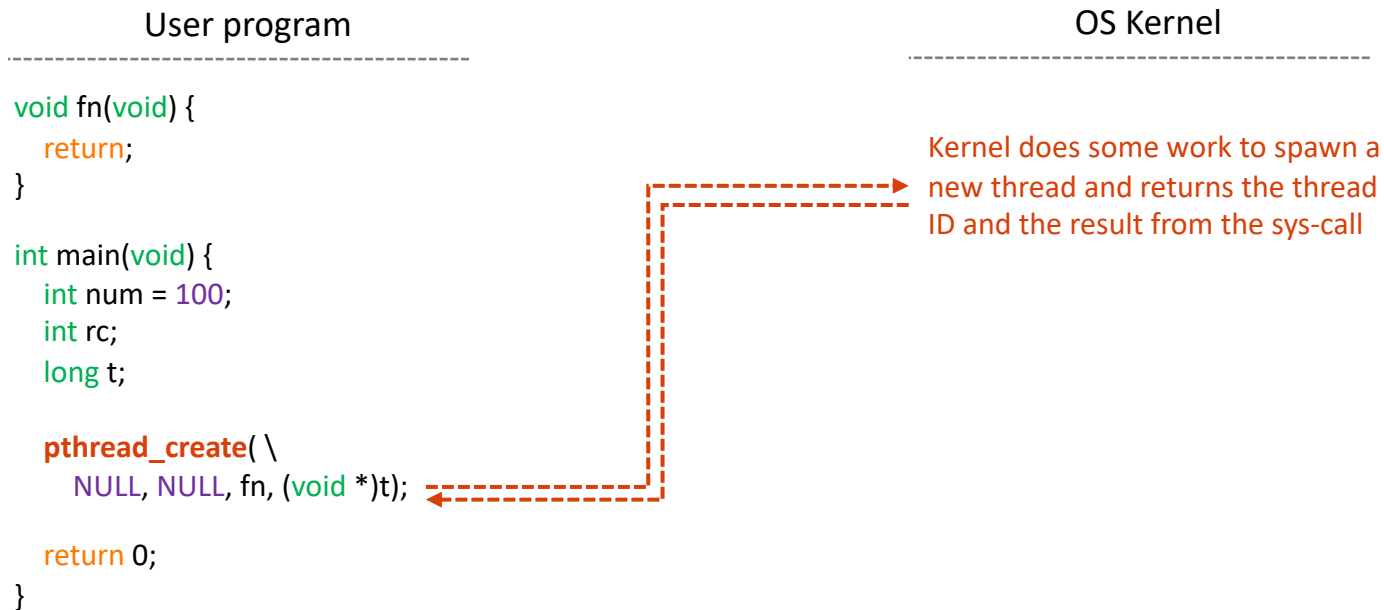
- How do we run a thread?
 - **System calls**
 - OS provide a set of system calls to control thread execution

OFFER STANDARD INTERFACE: **THREAD-SPECIFIC** SYSTEM CALLS

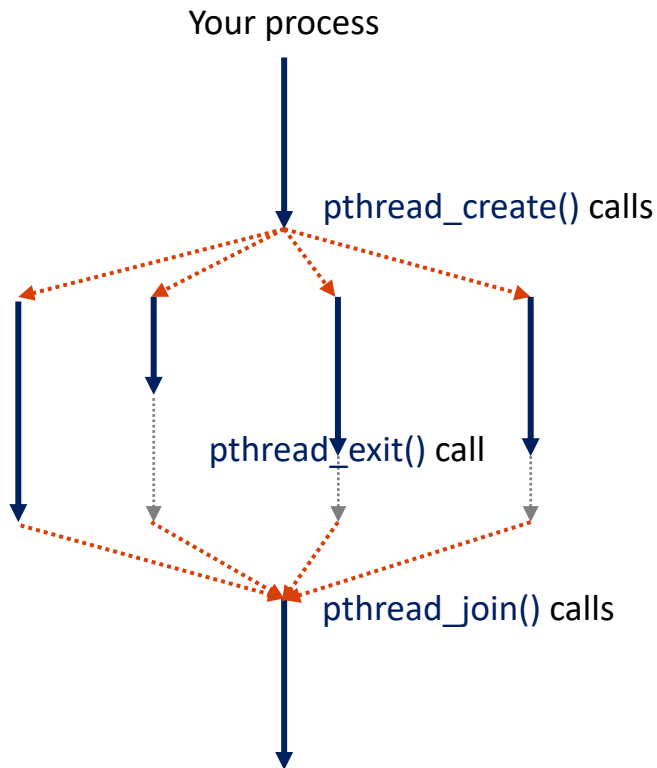
- Thread-specific system calls
 - **pthread_create**(thread, attribute, subroutine, subroutine-arguments);
 - Create a new thread executing the *subroutine* in the current process
 - Returns zero if it's successful, otherwise it returns [errno](#)
 - **pthread_exit**(return-value);
 - Terminate the thread and returns the *return-value* to any successful join
 - Note: If a thread terminates, it will be automatically called and always return success
 - **pthread_join**(thread, return-value-loc);
 - Suspend execution of the calling thread until the *thread* terminates
 - Once the thread terminates, the function will copy the return value to *return-value-loc*
 - Returns zero if it's successful, otherwise it returns an error

OFFER STANDARD INTERFACE: **THREAD-SPECIFIC** SYSTEM CALLS

- System call
 - **Example:** `pthread_create(...)`



THREAD PROGRAMMING: FORK-JOIN PATTERN



- **Fork - Join** Pattern
 - **Fork**: Main process creates a set of sub-(or child)-threads that runs a function
 - Each thread exits if the function returns
 - **Join**: Main waits until all the threads exit
- **Example**: download a large file
 - Splits a file into smaller chunks
 - Create a thread for downloading each
 - Sum-up all the downloaded chunks and combine them to create a single large file

OFFER STANDARD INTERFACE: **THREAD-SPECIFIC** SYSTEM CALLS

- Thread sample code in C
 - How many threads are there?
 - Which thread is created first?
 - Which thread is created last?
 - Which thread runs first/last?
 - What'd be an order of thread joins?
 - What will happen if we run this again?

```
static int value = 128;

void *subroutine (void *threadid) {
    long tid = (long) threadid;
    printf("Thread ID [%lx], value [%d]\n", tid, value ++);
}

int main(int argc, char *argv[]) {
    long t;
    int nthreads = 3;

    pthread_t *threads = (pthread_t *) malloc(nthreads * sizeof(pthread_t));
    memset(threads, 0x00, nthreads * sizeof(pthread_t));

    for (t = 0; t < nthreads; t++) {
        int rc = pthread_create(&threads[t], NULL, subroutine, (void *)t);
        if (rc) {
            printf("[Error] return code is: %d, abort.\n", rc);
            exit(-1);
        }
    }

    for (t = 0; t < nthreads; t++)
        pthread_join(threads[t], NULL);

    return 0;
}
```

OFFER STANDARD INTERFACE: **THREAD-SPECIFIC** SYSTEM CALLS

- Thread sample code in C
 - How many threads are there?
 - Which thread is created first?
 - Which thread is created last?
 - Which thread runs first/last?
 - What'd be an order of thread joins?
 - What will happen if we run this again?

Possible execution result:

Thread ID [0], value [128]

Thread ID [2], value [129]

Thread ID [1], value [130]

```
static int value = 128;

void *subroutine (void *threadid) {
    long tid = (long) threadid;
    printf("Thread ID [%lx], value [%d]\n", tid, value++);
}

int main(int argc, char *argv[]) {
    long t;
    int nthreads = 3;

    pthread_t *threads = (pthread_t *) malloc(nthreads * sizeof(pthread_t));
    memset(threads, 0x00, nthreads * sizeof(pthread_t));

    for (t = 0; t < nthreads; t++) {
        int rc = pthread_create(&threads[t], NULL, subroutine, (void *)t);
        if (rc) {
            printf("[Error] return code is: %d, abort.\n", rc);
            exit(-1);
        }
    }

    for (t = 0; t < nthreads; t++)
        pthread_join(threads[t], NULL);

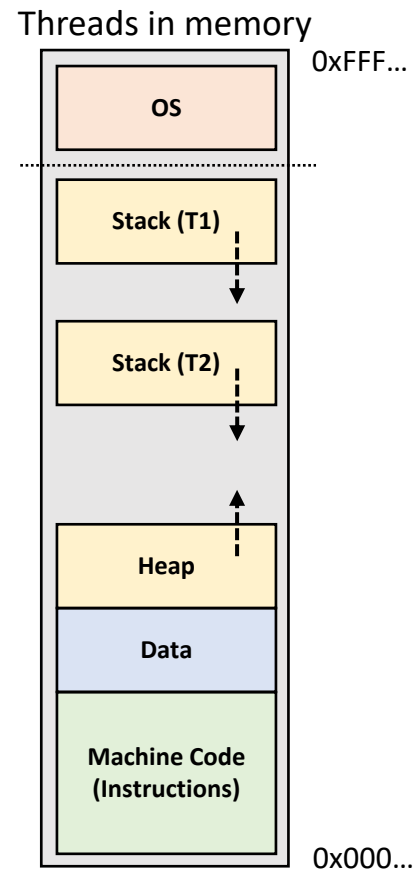
    return 0;
}
```

OFFER STANDARD INTERFACE: HOW OS MANAGES THREADS?

- **(Linux) OS**
 - A thread is treated as the same as a process
 - (Linux) thread control block \approx process context
- A thread can have **three states**:
 - **Ready**: a thread is created and ready to run, but not running now
 - **Running**: a thread running now
 - **Blocked**: a thread is unable to run (terminated or errors)

OFFER STANDARD INTERFACE: HOW OS MANAGES THREADS?

- Mem layout with two threads
 - Each thread has its own stack
 - Data, code and heap are shared between the two



TOPICS FOR TODAY

- Part I: Threads
 - Provide abstraction
 - What is a thread?
 - How is it different from a process?
 - How does OS run threads?
 - Offer standard libraries
 - How do we create/run/kill a thread?
 - How does OS manage the thread(s) we ran?
 - Manage resources
 - (Note) We will talk about this in the “scheduling” and “synchronization” classes

Thank You!

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu



Oregon State
University

SAIL
Secure AI Systems Lab