CS 344: Operating Systems I
01.23: Threads

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong
sanghyun.hong@oregonstate.edu

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?

```
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) {
   int apple_stock = 99;

   printf("Google stock price is %d\n", google_stock);
   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);
   printf("Apple   stock price is %d\n", ++apple_stock);

   return 0;
}
```

Run

Run

Run

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?

```
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?

```
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?

```
        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)

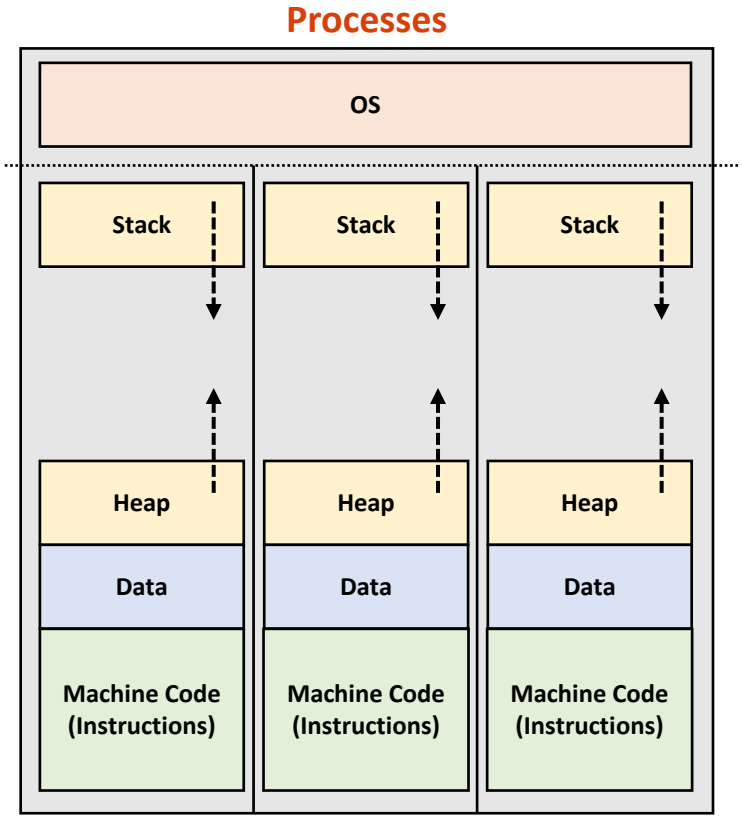
This procedure will be the **same** for all users
> **Multi-process** web-server

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)

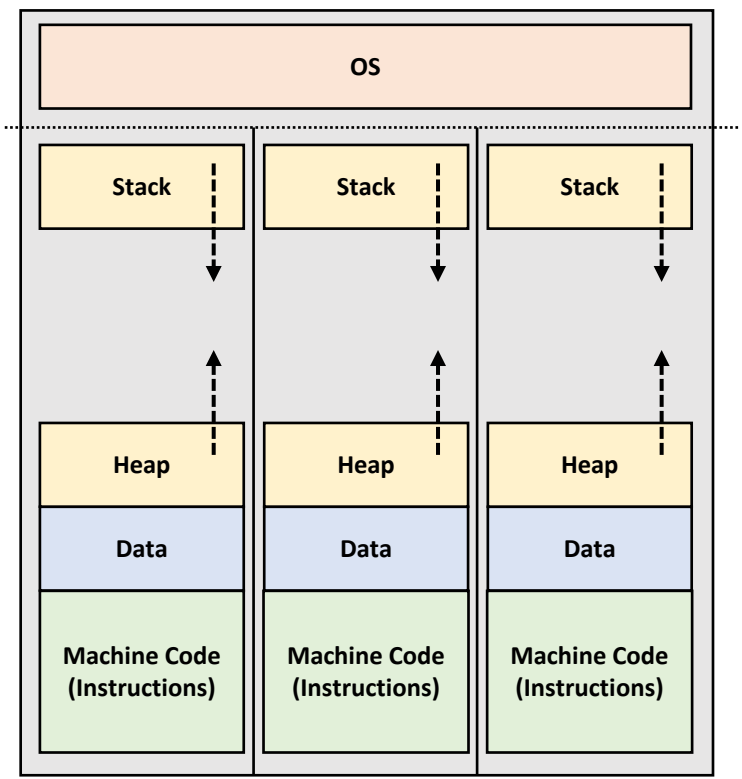


MULTI-PROCESS WEB-SERVER EXAMPLE: POTENTIAL ISSUES

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

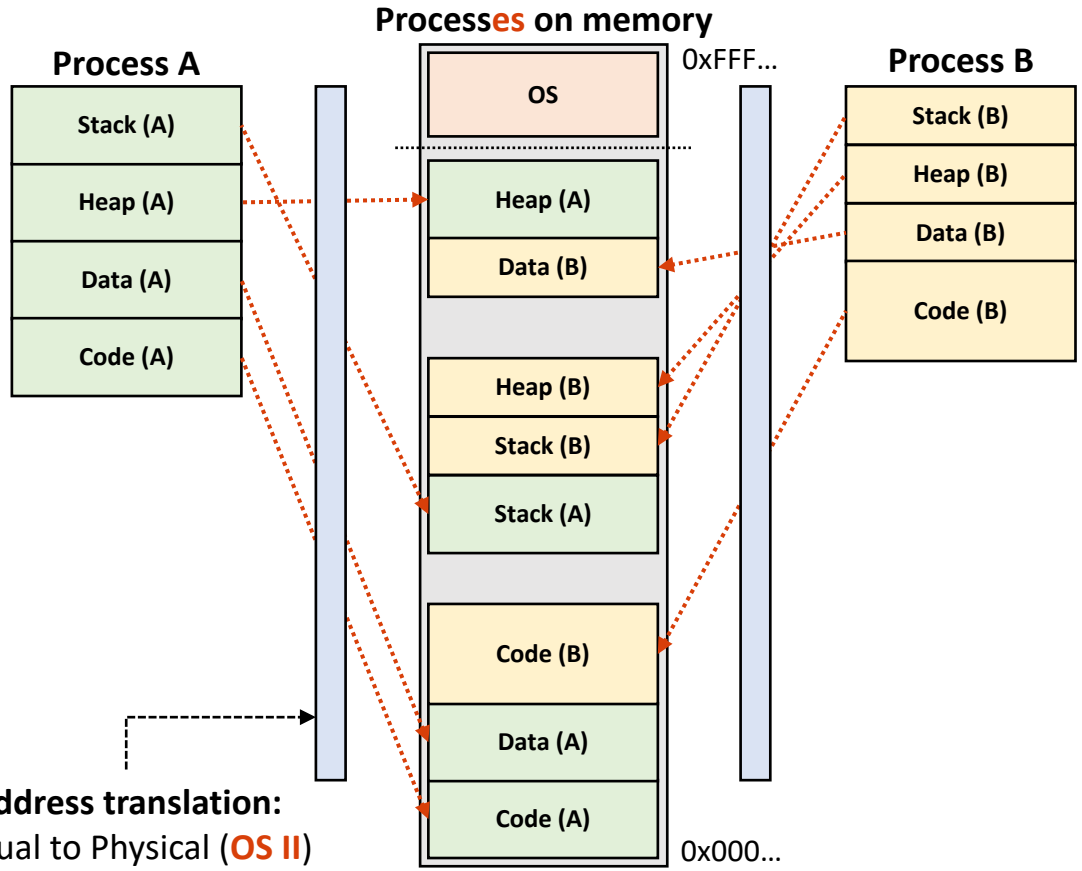
```
int main(void) {

  // 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!**



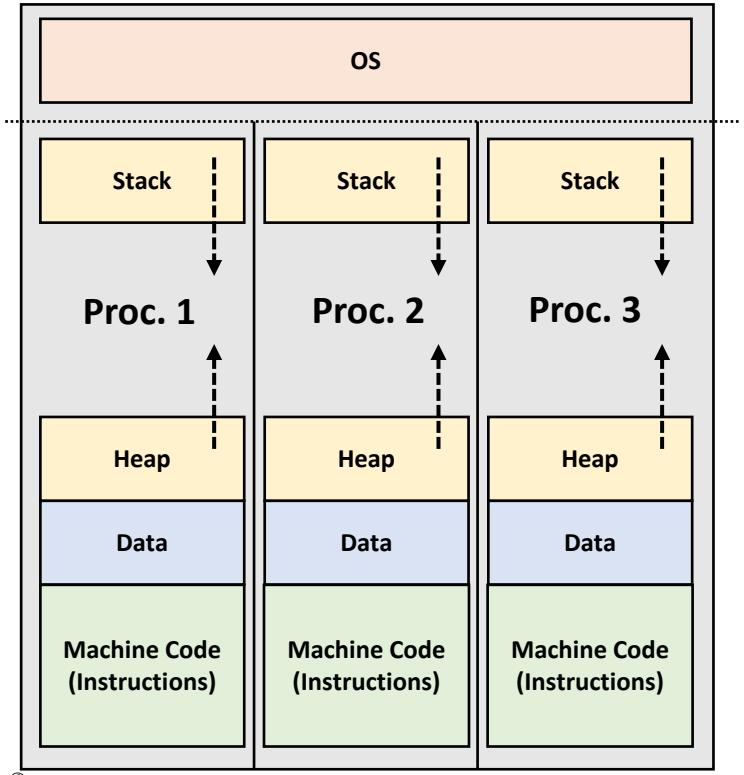
Solution: threads

Provide abstraction: a thread

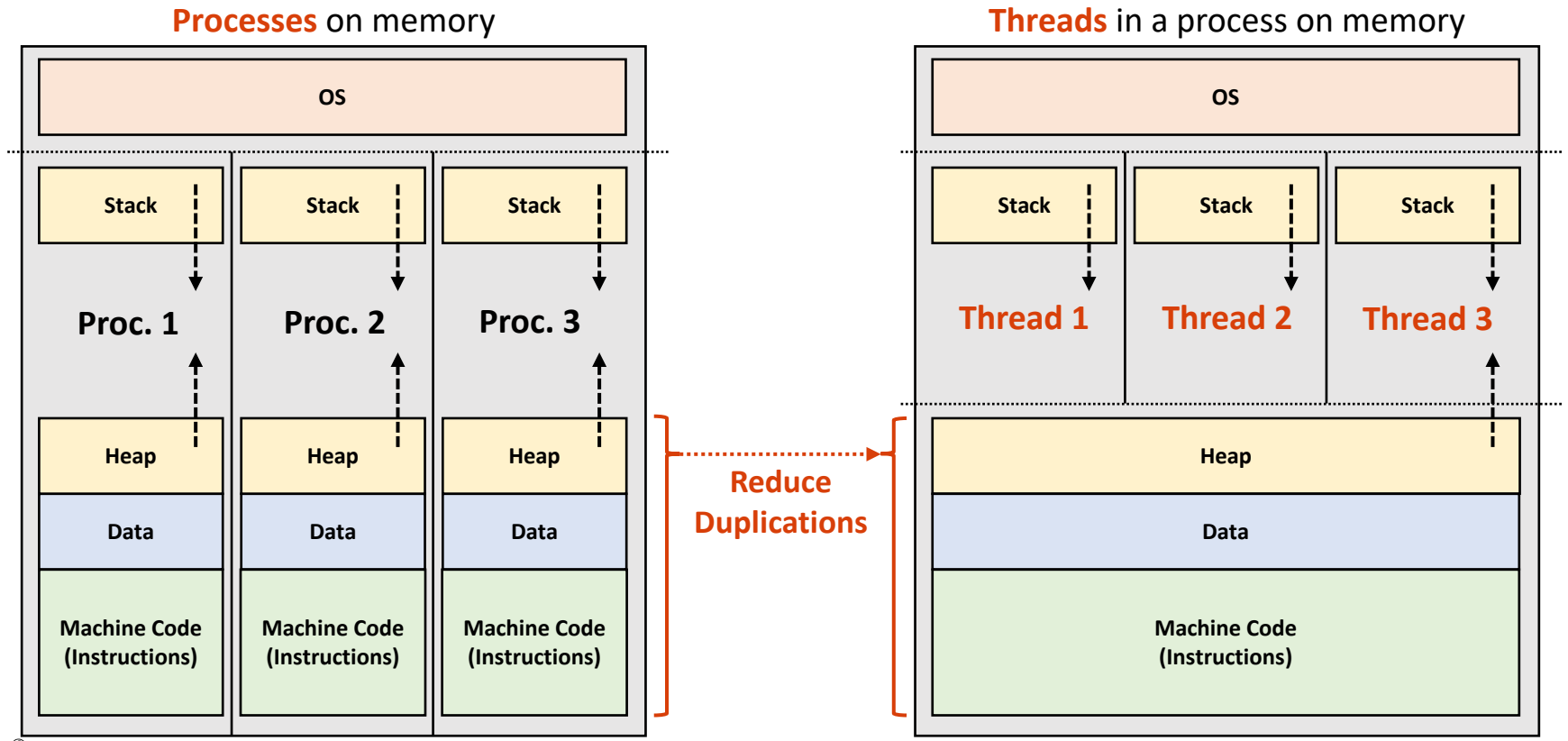
- Thread
  - **Definition:** a smallest schedulable execution context
  - **Terminology:**
    - Smallest: it's much light-weight than a process
    - Schedulable execution context: one thread can run on a CPU at a time

Provide abstraction: a thread – cont'd

Processes on memory



Provide abstraction: a thread – cont'd

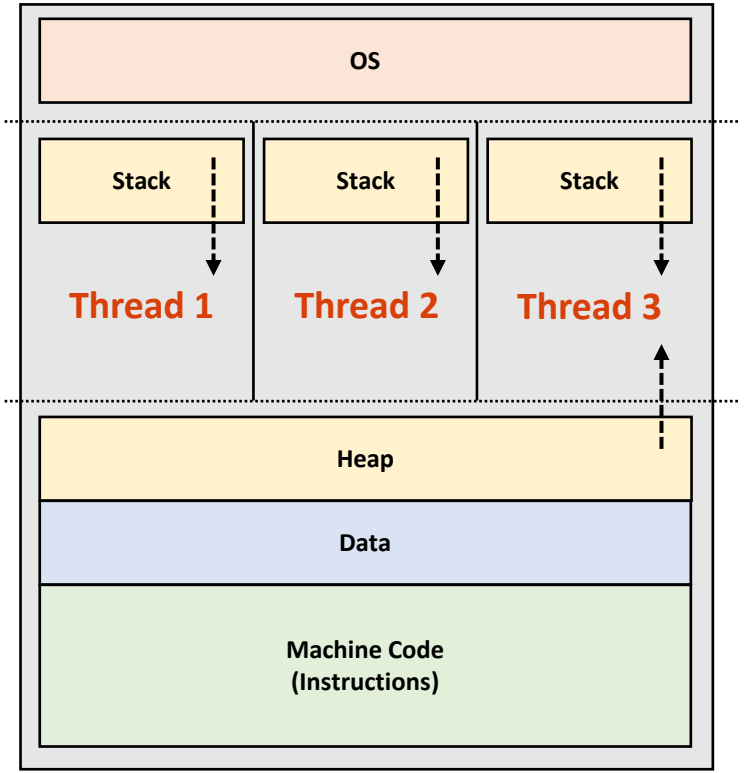


Provide abstraction: how is it different from a process?

- 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
    - …

Threads in a process on memory



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;
```

```
852         struct sched_info               sched_info;
853
854         struct list_head                tasks;
855 #ifdef CONFIG_SMP
856         struct plist_node               pushable_tasks;
857         struct rb_node                  pushable_dl_tasks;
858 #endif
859
860         struct mm_struct                *mm;
861         struct mm_struct                *active_mm;
862
863         /* Per-thread vma caching: */
864         struct vmacache                 vmacache;
865
866 #ifdef SPLIT_RSS_COUNTING
867         struct task_rss_stat            rss_stat;
868 #endif
869         int                             exit_state;
870         int                             exit_code;
871         int                             exit_signal;
872         /* The signal sent when the parent dies: */
873         int                             pdeath_signal;
874         /* JOBCTL_*, siglock protected: */
875         unsigned long                   jobctl;
```

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

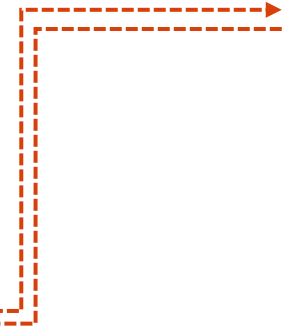
- System call
  - **Example:** pthread_create(…)

User program

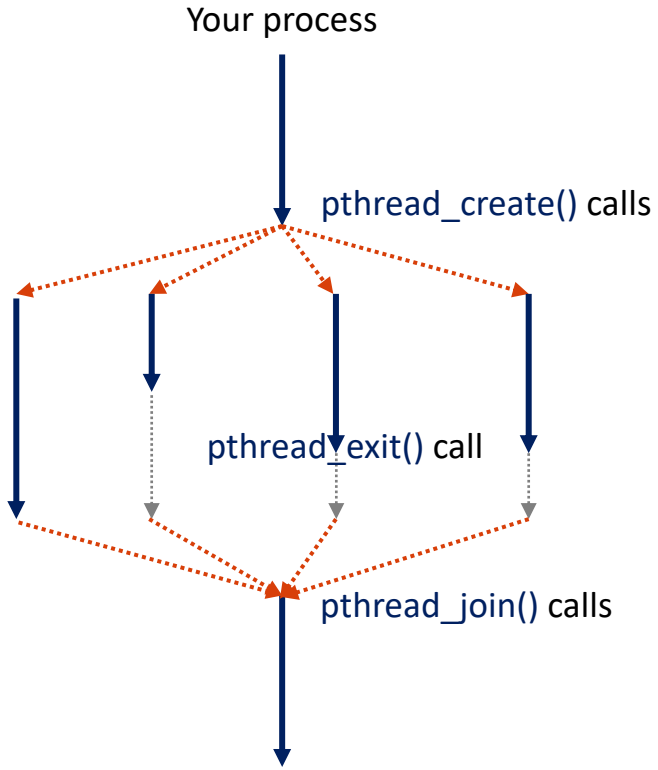
```
void fn(void) {
   return;
}

int main(void) {
   int num = 100;
   int rc;
   long t;

   pthread_create( \
      NULL, NULL, fn, (void *)t);

   return 0;
}
```

OS Kernel

Kernel does some work to spawn a new thread and returns the thread ID and the result from the sys-call



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]

```c
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;
}
```

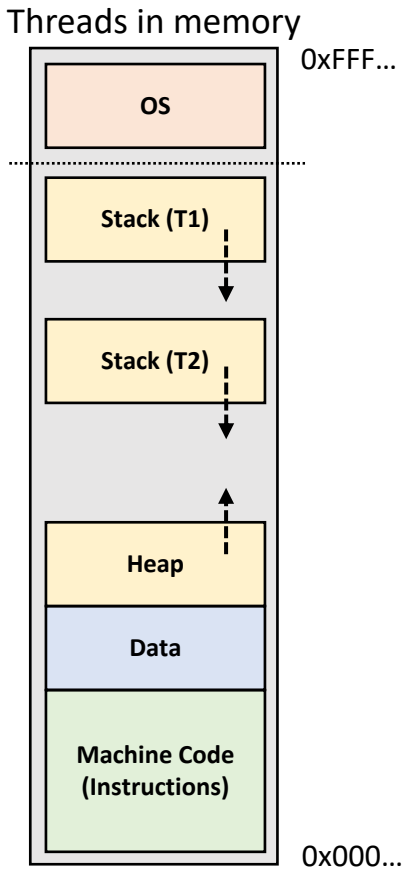
Offer standard interface: how OS manages threads?

- **(Linux)** OS
  - A thread is treated as the same as a process
  - (Linux) thread control block ≈ 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