NOTICE – CONT'D

- Deadlines
  - (4/13 11:59 PM) Programming assignment 1
  - (4/20 11:59 PM) Midterm quiz 1

RECAP – CONT'D

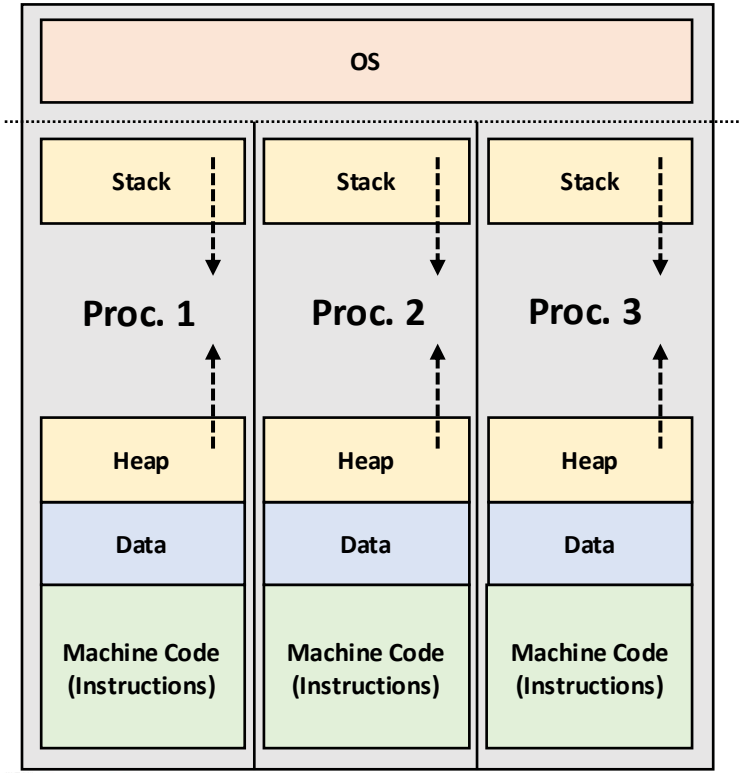
- Part I: Threads
  - Provide abstraction
    - What is a thread?
    - How is it different from a process?
    - How does OS run threads?
  - Offer standard libraries
    - How do we create/run/kill a thread?
    - How does OS manage the thread(s) we ran?
  - Manage resources
    - (Note) We will talk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

PROVIDE ABSTRACTION: A THREAD

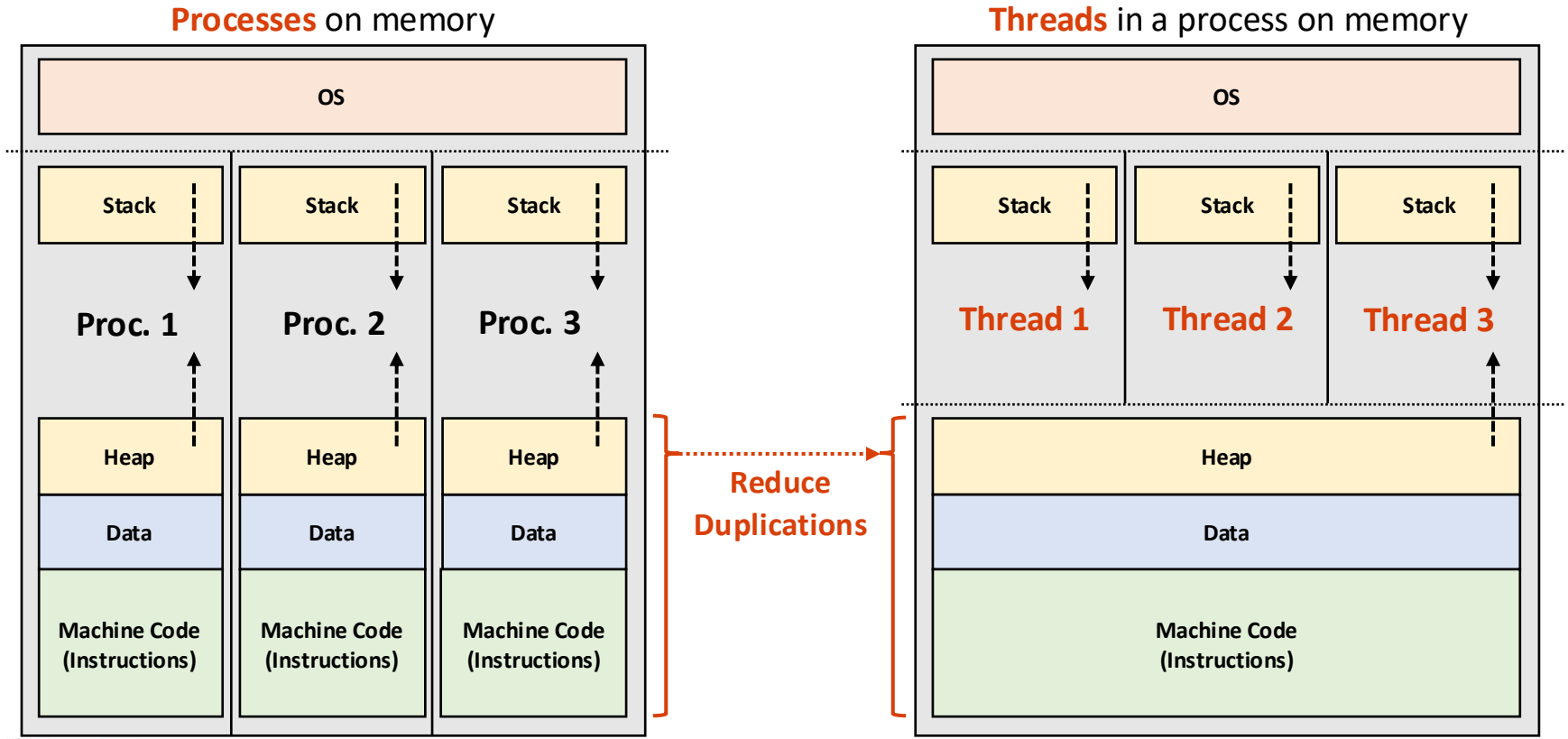
- Thread
  - **Definition:** a smallest schedulable execution context
  - **Terminology:**
    - Smallest: it's much light-weight than a process
    - Schedulable execution context: one thread can run on a CPU at a time

PROVIDE ABSTRACTION: A THREAD – CONT'D

Processes on memory



PROVIDE ABSTRACTION: A THREAD – CONT'D

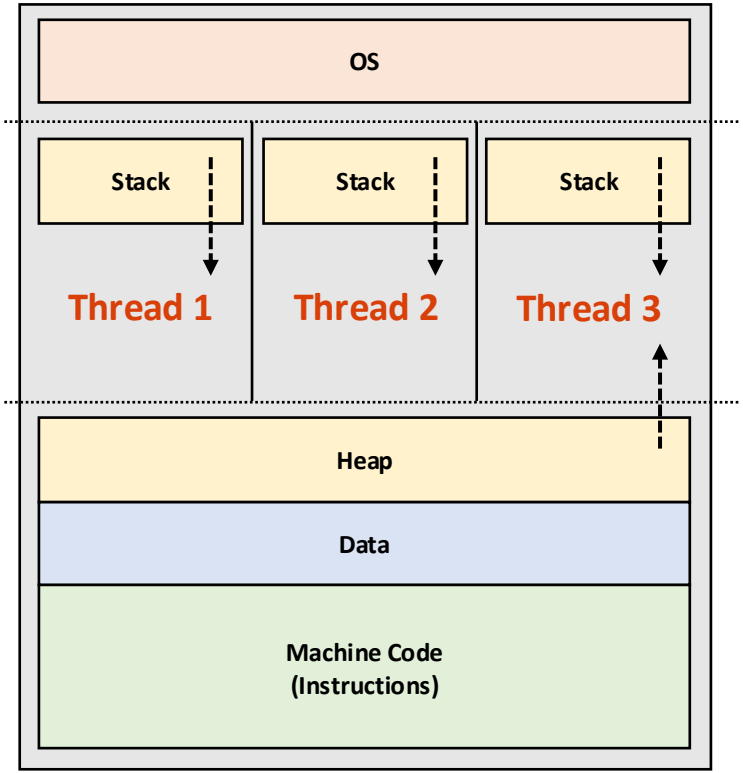


PROVIDE ABSTRACTION: HOW IS IT DIFFERENT FROM A PROCESS?

- 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

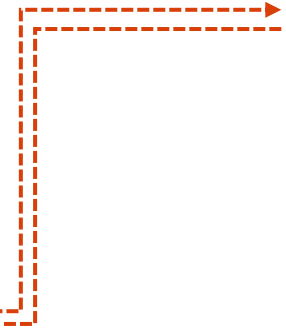
- System call
  - **Example:** pthread_create(…)

User program

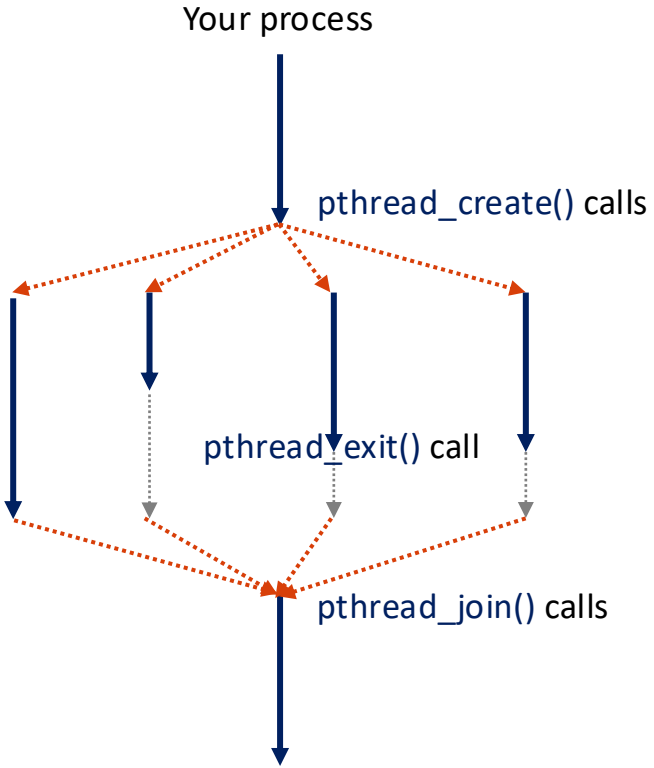
```
void fn(void) {
   return;
}

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?

```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: 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;
}
```

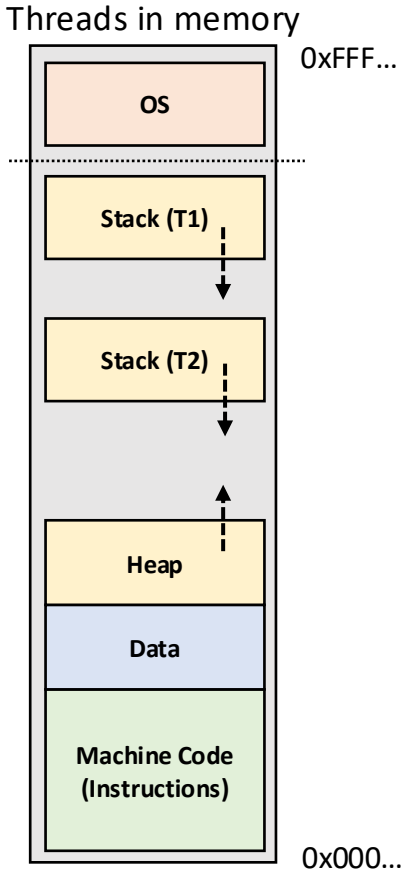
OFFER STANDARD INTERFACE: HOW OS MANAGES THREADS?

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



CS 374: Operating Systems I

Part I – Context switching

M/W 12:00 – 1:50 PM (LINC #200)

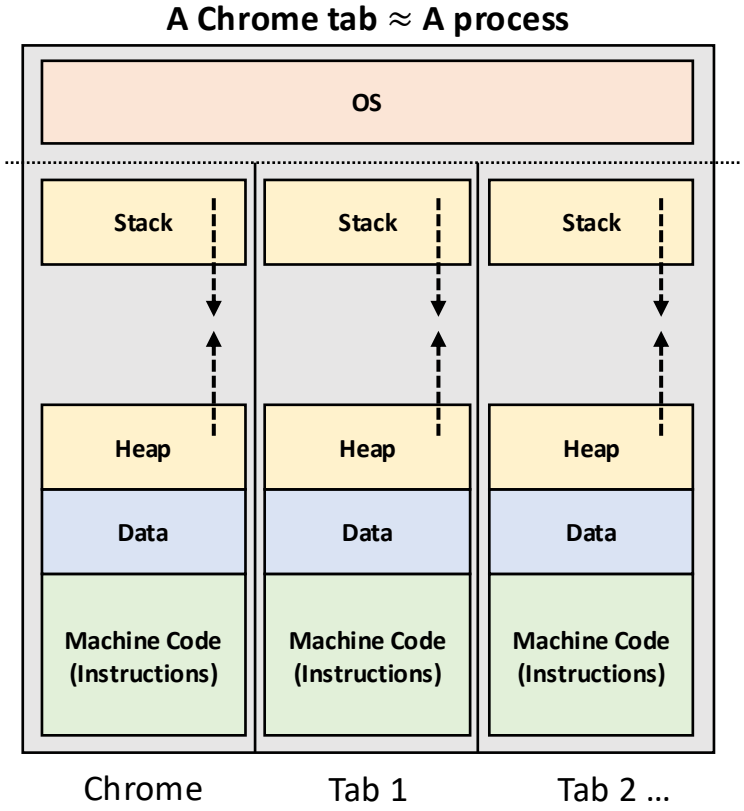
Sanghyun Hong

sanghyun.hong@oregonstate.edu

PROBLEM: MULTIPLE PROGRAM, YET LIMITED PROCESSORS

- Your Chrome browser:
  - Open multiple websites (tabs)
    - Tab 1: Open Canvas website
    - Tab 2: Stack Overflow
    - Tab 3: Discord website
    - … (many more 10+)
  - 4-8 CPUs (Processors)

How Can OS Address This Problem?



PROBLEM: MULTIPLE PROGRAM, YET LIMITED PROCESSORS – CONT'D

- Solution: (process) scheduling
  - **Goal:**
    - generate illusion that you're running 100+ processes at the same time
    - But in truth, it's not

SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term..**:

  - **Scenario:** open a website for downloading movies

SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term..**:

  - **Scenario:** the website opened and open two other websites

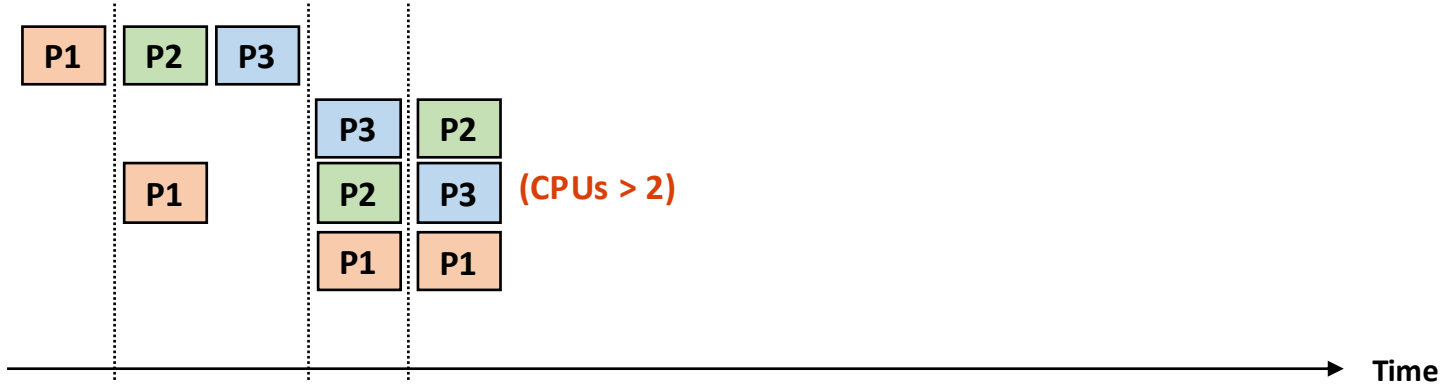
SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term.**.:

  - **Scenario:** downloads started, and you focus on Canvas

SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

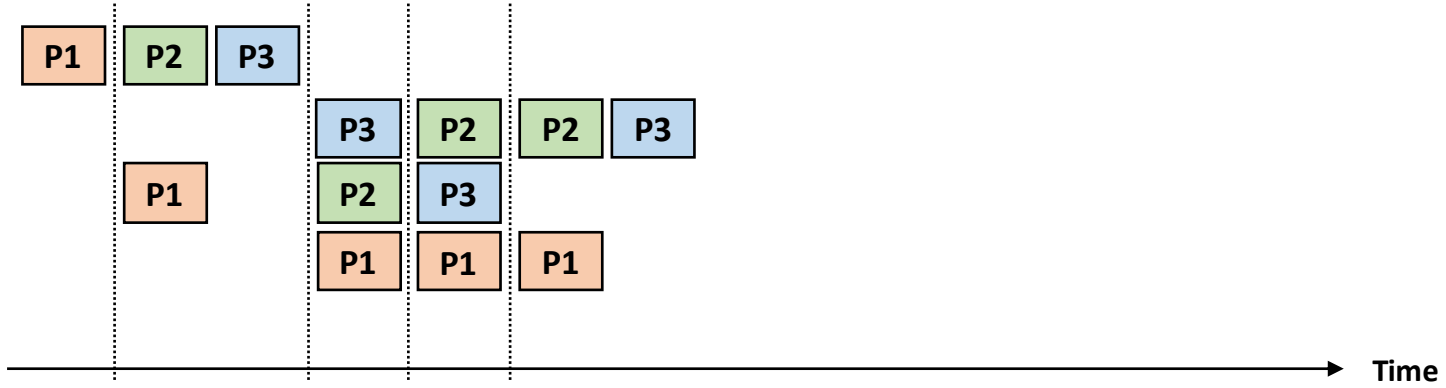
- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term..**:



  - **Scenario:** while downloading, you start searching StackOverflow

SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

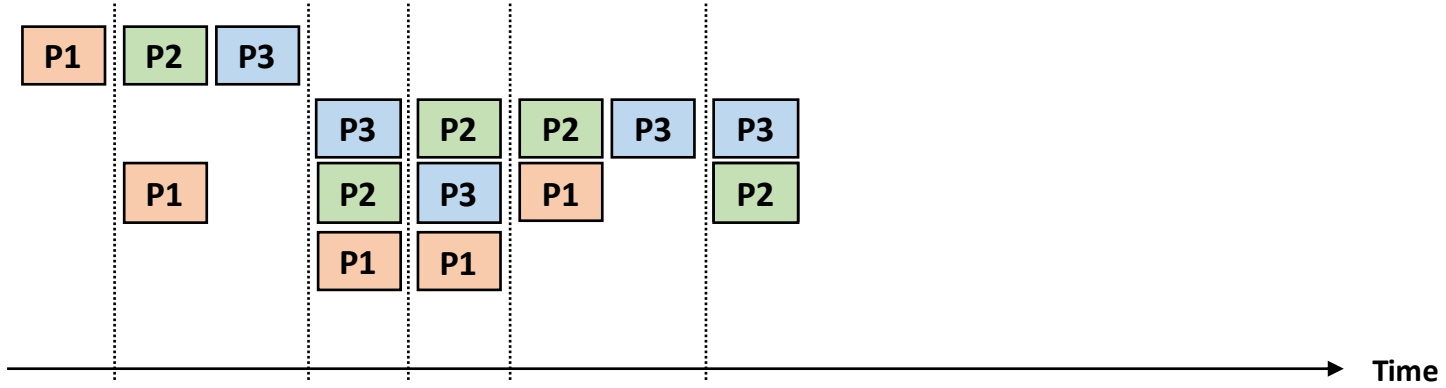
- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term..**:



  - **Scenario:** downloading movies are done

SCHEDULING EXAMPLE: HIGH-LEVEL VIEW

- 3 Processes in Chrome:
  - **P1:** Download movies
  - **P2:** Open Canvas
  - **P3:** Search StackOverflow
- Example
  - **New** :
  - **Ready**:
  - **Run** :
  - **Wait** :
  - **Term..**:



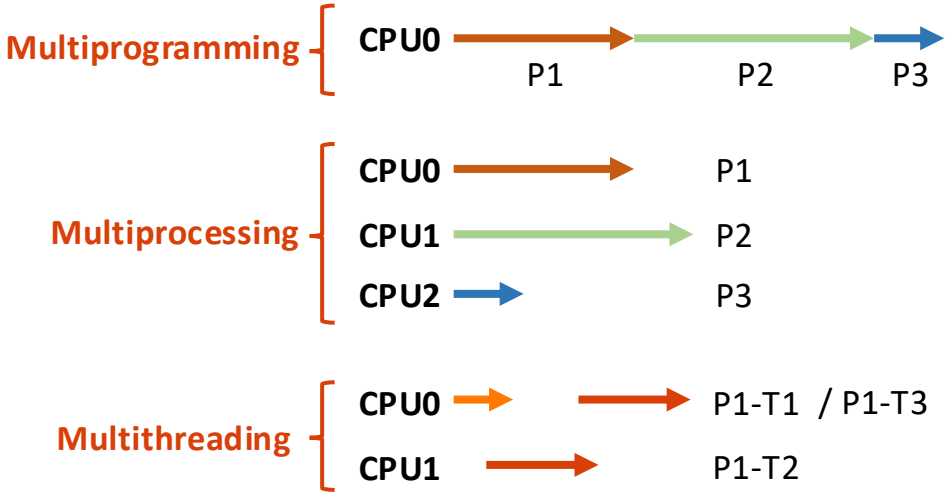
  - **Scenario:** close the download tab, and keep looking at Canvas

TOPICS FOR TODAY

- Part I: Context switching
  - Provide abstraction
    - Why do we need context switching?
    - What is a context?
    - How does OS perform a context switch?
  - Offer standard libraries
    - How do we observe a context switch?
    - How do we trigger a context switch manually?
    - How do we save and restore a context ourselves?
  - Manage resources
    - (Note) We will talk “who gets the GPU next” in the scheduling class
    - (Note) We will talk about “shared resource conflicts” in the synchronization class

PRELIMINARIES ON TERMINOLOGY

- **Definitions:**
  - Multiprogramming vs. multi-processing vs. multi-threading
    - Multi-programming: multiple jobs (or processes)
    - Multi-processing: multiple processors (CPUs)
    - Multi-threading: multiple threads



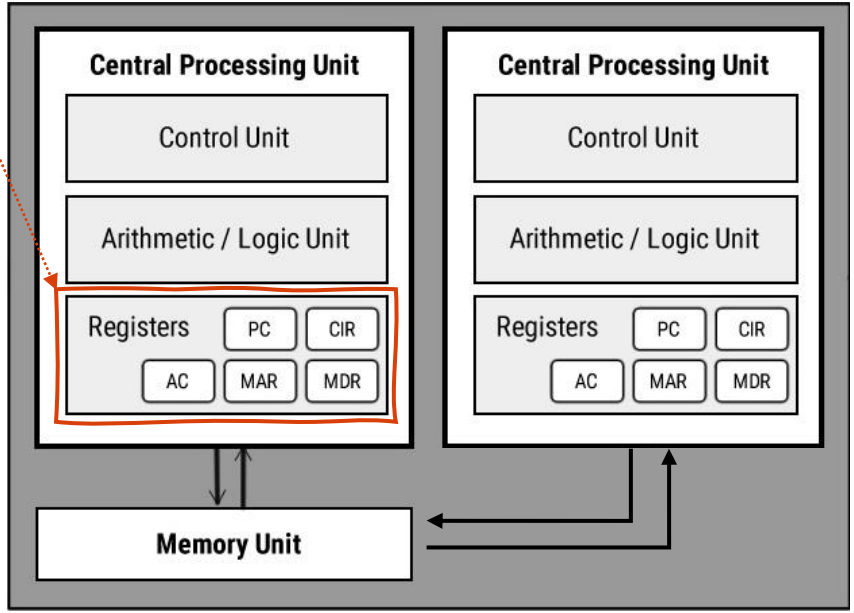
WHAT IS CONTEXT SWITCHING?

- **Context switching**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another

WHAT IS A (PROCESS/THREAD) CONTEXT?

- (Linux) Process context in x86 Intel CPU architectures
  - **Code**
    - Program counter (PC)
    - Instruction pointer (EIP)
  - **Stack and heap**
    - Stack pointer (ESP, EBP)
    - Heap pointer (Page table)
  - **Running context**
    - Process state (ID, …)
    - Execution flags
    - CPU # to run
    - Scheduling policy
    - Mem. virtualization
  - …

Process Context: A set of information that OS requires to run a process on a CPU, different from CPU vendors (ex. In Linux, it's defined as *task_struct*, Link)



WHAT IS A (PROCESS/THREAD) CONTEXT?

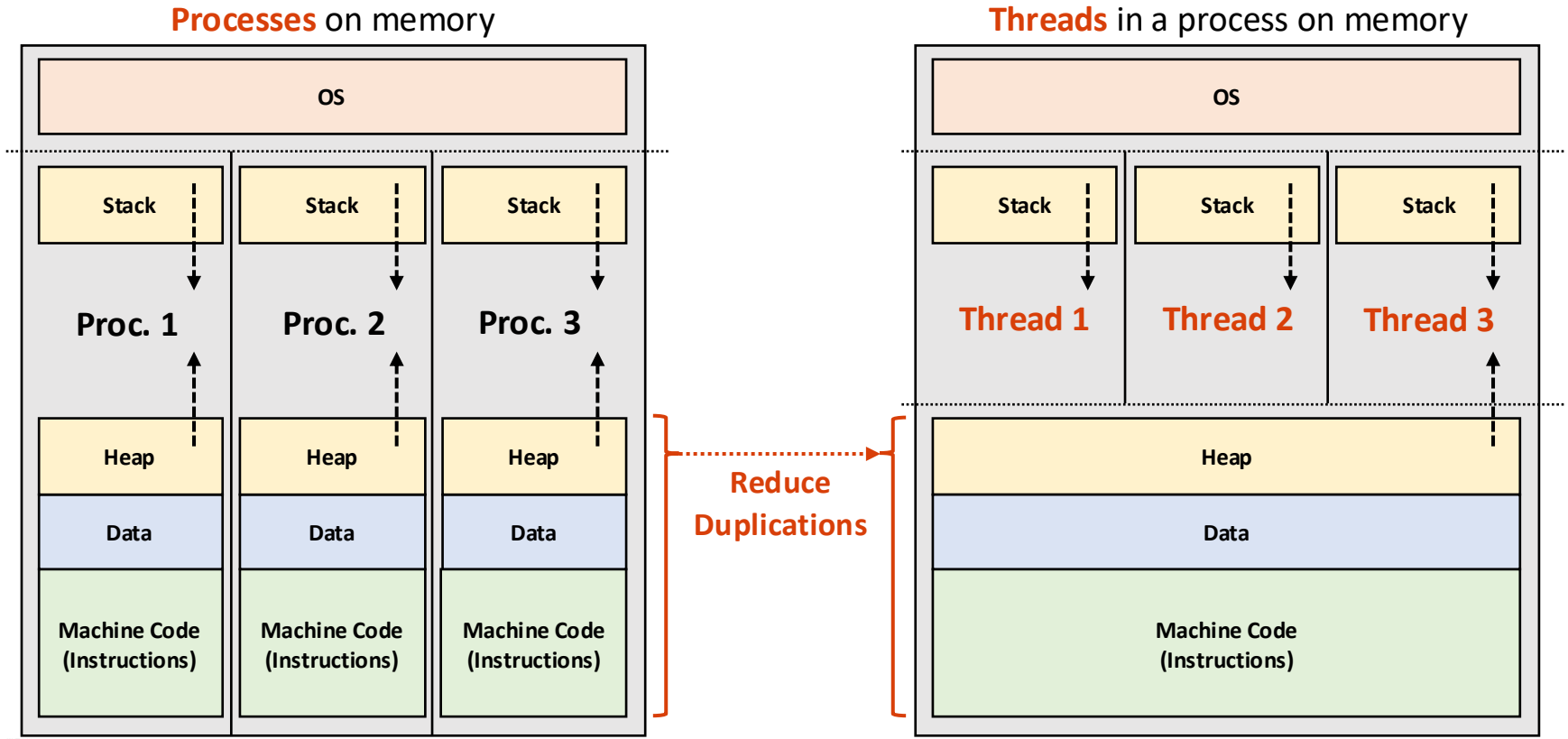
- (Linux) defines this process context
  - **Code**
    - Program counter
    - Instruction pointer
  - **Stack and heap**
    - Stack pointer
    - Heap pointer
  - **Running context**
    - Process state (ID, …)
    - Execution flags
    - CPU # to run
    - Scheduling policy
    - Mem. virtualization
  - …

Process Context: A set of information that OS requires to run a process on a CPU, different from CPU vendors (ex. In Linux, it's defined as *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;
876
877         /* Used for emulating ABI behavior of previous Linux versions: */
878         unsigned int                    personality;
```

RECAP: PROCESS VS. THREAD



WHAT IS A (PROCESS/THREAD) CONTEXT?

- (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
    - Scheduling policy
    - 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)

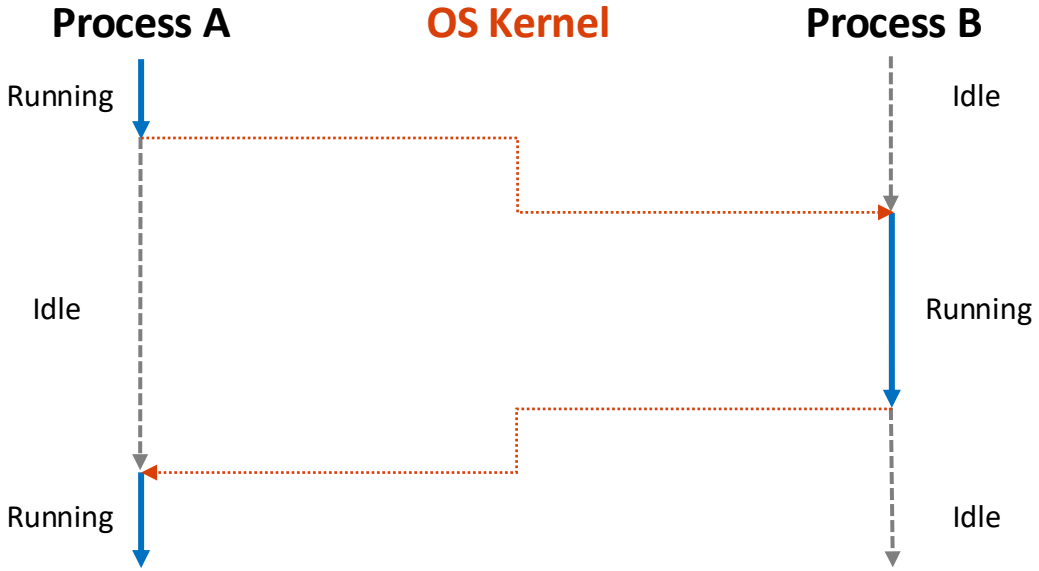
```
728 struct task_struct {
729 #ifdef CONFIG_THREAD_INFO_IN_TASK
730         /*
731          * For reasons of header soup (see current_thread_info()), this
732          * must be the first element of task_struct.
733          */
734         struct thread_info              thread_info;
735 #endif
736         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

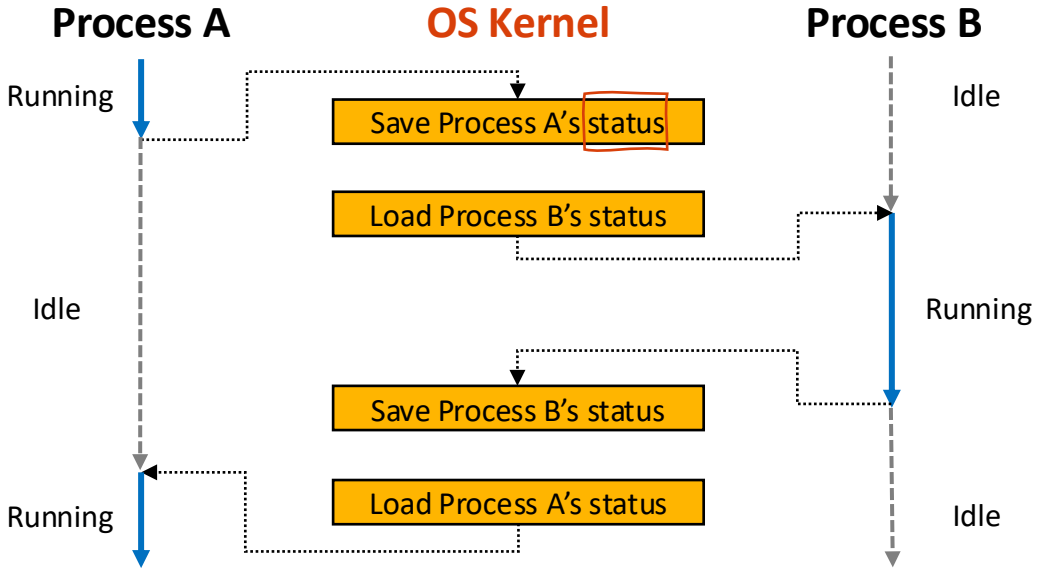
How does OS perform context switching?

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



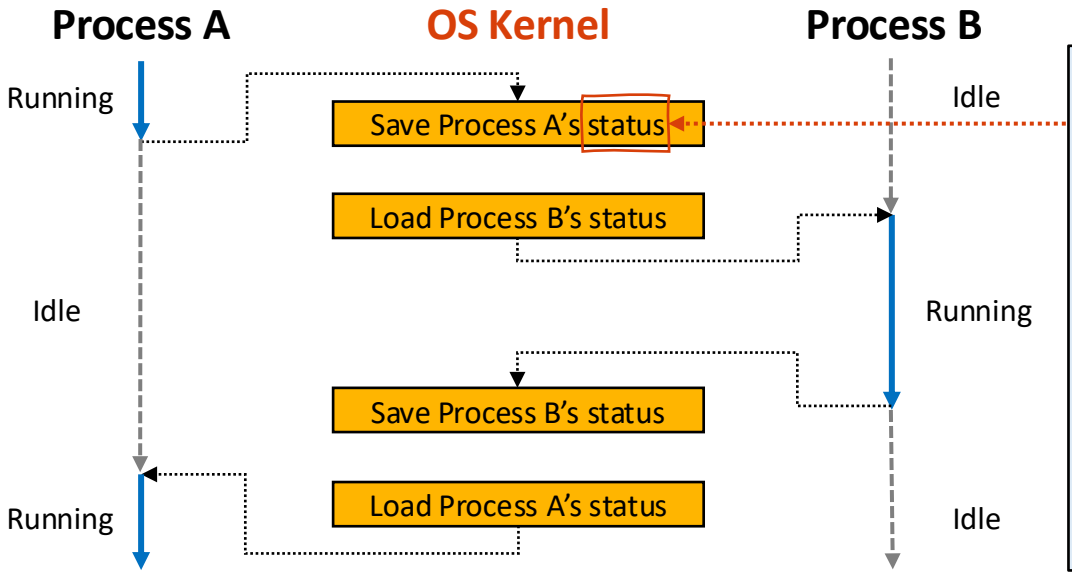
How does OS perform context switching?

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



How does OS perform context switching?

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another



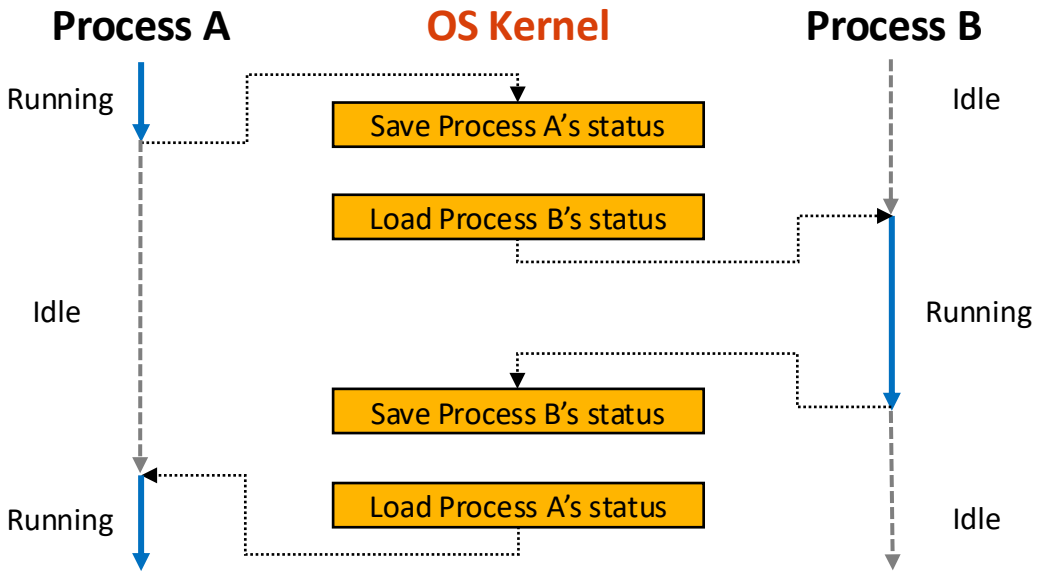
Recall: Process control block

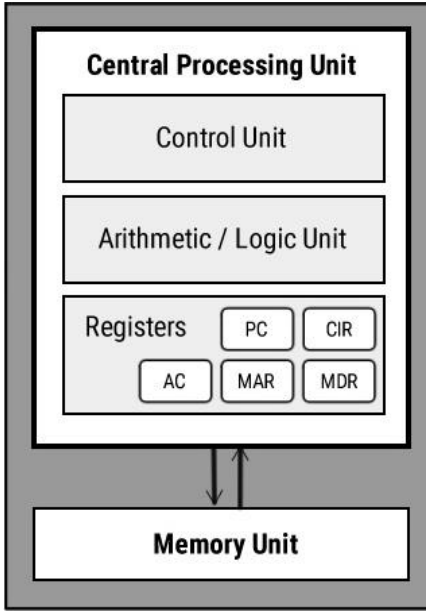
A structure in OS that contains a set of information required to run a process on a CPU. Recall that Linux has *task_struct*.

- CPU#
- Program counter
- Instruction pointer
- Heap/stack pointer
- Process state [!]
- …

How does OS perform context switching?

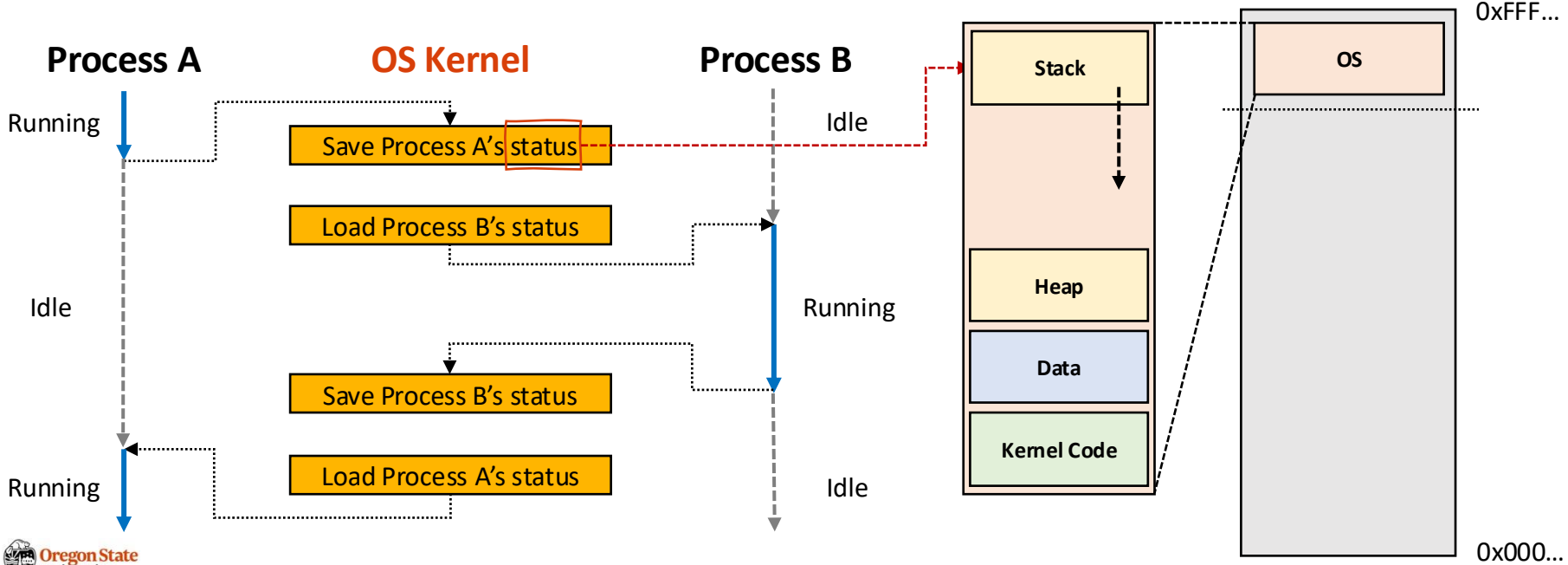
- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another





WHERE DOES OS STORE THE CONTEXT?

- **OS kernel**
  - **Definition:** a program that has a complete control over the system
  - OS kernel stores the context in its own *stack (not in the user stack)*



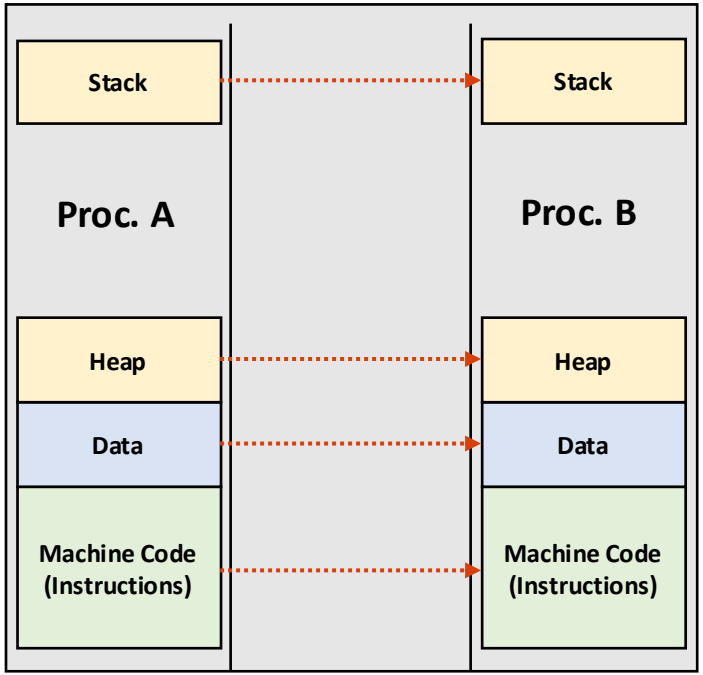
How does OS perform context switching?

- **Context switch**
  - **Definition:** OS stores the current process's status and loads the new process's one
  - **Informal:** OS takes a CPU from one process and gives it to another

- No free lunch
  - Context switching takes $\sim$5 μs on average
  - OS typically runs 100+ processes
  - Too many context switching makes a system unable to respond…

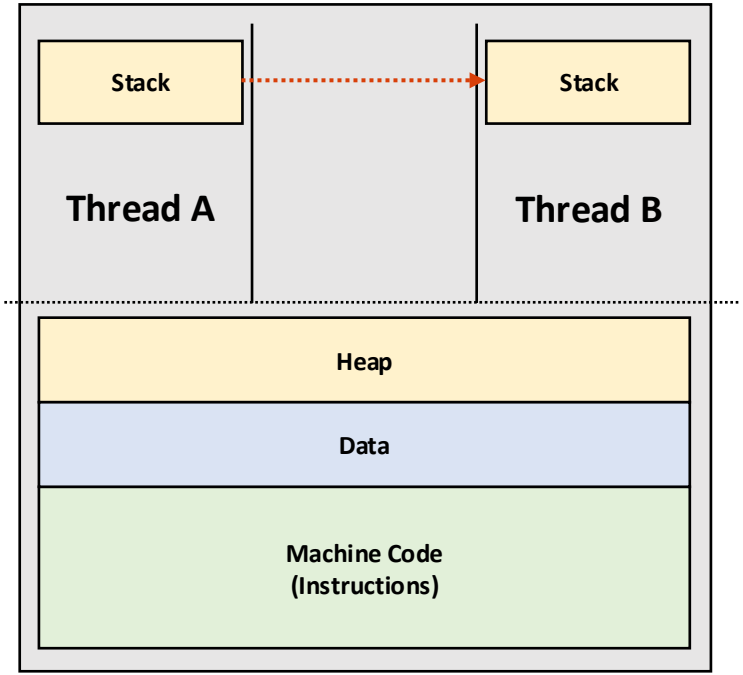
PROCESS VS. THREAD CONTEXT SWITCHING

Process switch



Process switch needs to replace *all*
(4 segments, registers… – needs TLB flush)

Thread switch



Thread switch needs to replace *only*
registers and stack – lightweight

TOPICS FOR TODAY

- Part I: Context switching
  - Provide abstraction
    - Why do we need context switching?
    - What is a context?
    - How does OS perform a context switch?
  - Offer standard libraries
    - How do we observe a context switch?
    - How do we trigger a context switch manually?
    - How do we save and restore a context ourselves?
  - Manage resources
    - (Note) We will talk “who gets the GPU next” in the scheduling class
    - (Note) We will talk about “shared resource conflicts” in the synchronization class

How do we observe a context switch?

- Process exposes its ctx switch count
  - /proc/[pid]/status
  - Example: $ cat /proc/$$/status | grep ctxt (check twice for difference)

- $$ = current (bash) shell process (I/O bound)
  - voluntary_ctxt_switches A
  - nonvoluntary_ctxt_switches B

- yes > /dev/null (Compute bound)
  - voluntary_ctxt_switches C
  - nonvoluntary_ctxt_switches D

How do we observe a context switch? – cont'd

- Trace system calls
  - strace -c program or strace -c -p [pid]
  - Example: $ strace -c cat /proc/$$/status

- Compare these
  - $$ = current (bash) shell process (I/O bound)
  - yes > /dev/null (Compute bound)

How do we trigger a context switch?

- Process can voluntarily give up the CPU
  - sched_yield() // in #include <sched.h>

- Kernel procedure when we run this example
  - Put the current thread to the thread *queue*
  - OS kernel selects the next thread
  - Note: if there is only a thread in the queue
    then we can observe only that thread running

```
void *thread_no_yield(void *arg) {
   int id = *(int *)arg;
   for (int i = 0; i < 5; i++) {
      printf("Thread %d: step %d\n", id, i);
   }
   return NULL;
}

void *thread_with_yield(void *arg) {
   int id = *(int *)arg;
   for (int i = 0; i < 5; i++) {
      printf("Thread %d: step %d\n", id, i);
      sched_yield();
   }
   return NULL;
}

int main() {
   pthread_t t1, t2;
   int id1 = 1, id2 = 2;
   printf("=== Without sched_yield ===\n");
   pthread_create(&t1, NULL, thread_no_yield, &id1);
   pthread_create(&t2, NULL, thread_no_yield, &id2);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);

   printf("\n=== With sched_yield ===\n");
   pthread_create(&t1, NULL, thread_with_yield, &id1);
   pthread_create(&t2, NULL, thread_with_yield, &id2);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
}
```

HOW DO WE TRIGGER A CONTEXT SWITCH? – CONT'D

- Process can voluntarily give up the CPU
  - sched_yield() // in #include <sched.h>
  - It increases "voluntary_ctxt_switches"

```
#include <stdio.h>
#include <sched.h>
#include <unistd.h>

int main() {
   pid_t pid = getpid();
   char cmd[100];

   printf("=== Before sched_yield ===\n");
   snprintf(cmd, sizeof(cmd),
      "cat /proc/%d/status | grep ctxt", pid);
   system(cmd);

  for (int i = 0; i < 100; i++) {
      sched_yield();
   }

   printf("\n=== After 100 x sched_yield ===\n");
   system(cmd);

   return 0;
}
```

How do we save and restore a context?

- POSIX supports
  - ucontext_t
  - A structure that stores *only the user stack*
- System calls
  - getcontext(ucp)
  - setcontext(ucp)
  - makecontext(ucp, fn, …)
  - swapcontext(oucp, ucp)

```
typedef struct ucontext_t {
    struct ucontext_t *uc_link;
    stack_t           uc_stack;
    mcontext_t        uc_mcontext;
    sigset_t          uc_sigmask;
 } ucontext_t;
```

```
ucontext_t ctx_main, ctx_worker;
char stack[65536];

void worker() {
  printf("Worker: step 1\n");
  swapcontext(&ctx_worker, &ctx_main);
  printf("Worker: step 2\n");
}

int main() {
  getcontext(&ctx_worker);
  ctx_worker.uc_stack.ss_sp = stack;
  ctx_worker.uc_stack.ss_size = sizeof(stack);
  ctx_worker.uc_link = &ctx_main;
  makecontext(&ctx_worker, worker, 0);

  printf("Main: before swap\n");
  swapcontext(&ctx_main, &ctx_worker);
  printf("Main: after first swap\n");
  swapcontext(&ctx_main, &ctx_worker);
  printf("Main: done\n");
}
```

TOPICS FOR TODAY

- Part I: Context switching
  - Provide abstraction
    - Why do we need context switching?
    - What is a context?
    - How does OS perform a context switch?
  - Offer standard libraries
    - How do we observe a context switch?
    - How do we trigger a context switch manually?
    - How do we save and restore a context ourselves?
  - Manage resources
    - (Note) We will talk “who gets the GPU next” in the scheduling class
    - (Note) We will talk about “shared resource conflicts” in the synchronization class

Thank You!

M/W 12:00 – 1:50 PM (LINC #200)

Sanghyun Hong

sanghyun.hong@oregonstate.edu