ECE391
Computer System Engineering
Lecture 17

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Fall 2022

Lecture Topics

- Programs to processes/tasks
  - Virtualization of a program -> a process
- Process/task identification and linkage
- Creating processes/tasks
- Task State Segment (TSS)

Aministrivia

- **MP3 Checkpoint 2**
  - **Due by 6:00pm Monday, October 24**

EXAM 2

Date: Tuesday, November 1

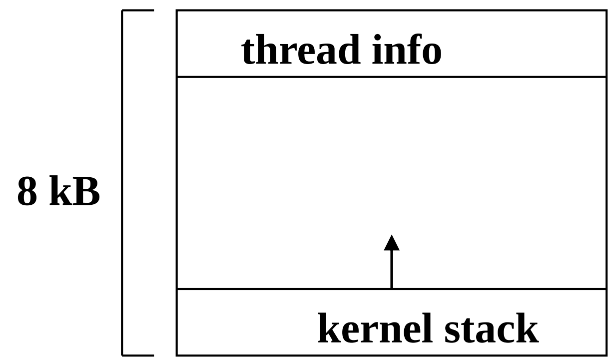
Time: 7:00pm - 9:00pm

Task Identification and Linkage

- What is a process/task?
  - unit of scheduling in Linux
  - also name of data structure (**`struct task_struct`**)
    (see linux/sched.h)
- User-level view
  - each execution context that can be independently scheduled must have its own process descriptor
  - traditional process id (pid, a field in task structure/process descriptor)
  - from 1 to 32,767 in Linux, used as task-unique identifier
  - ***tgid*** (thread group id) plays process id role for multithreaded applications (common id for all threads in process)
  - most processes belong to a thread group consisting of a single member

Task Identification and Linkage (cont.)

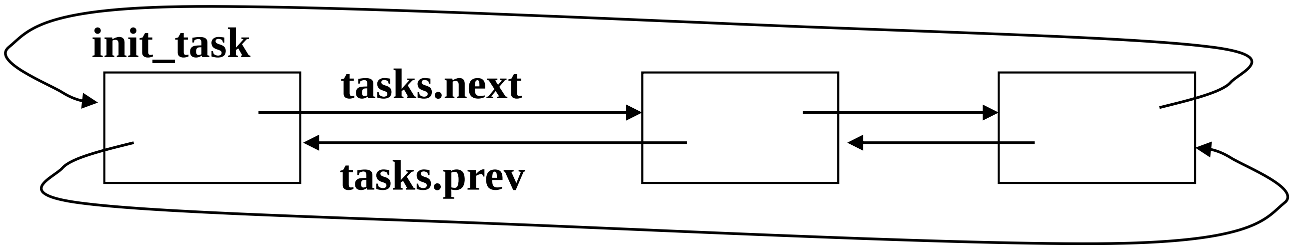
- Kernel view
  - kernel must handle many processes at the same time
  - keeps two data structures in a single per-process area (8kB)
    - `thread_info` structure (keeps pointer to task structure or process descriptor)
    - kernel stack
    - both dynamically allocated
  - architecture-dependent thread info shares space with kernel stack



- Remember: DO NOT USE RECURSION IN THE KERNEL!

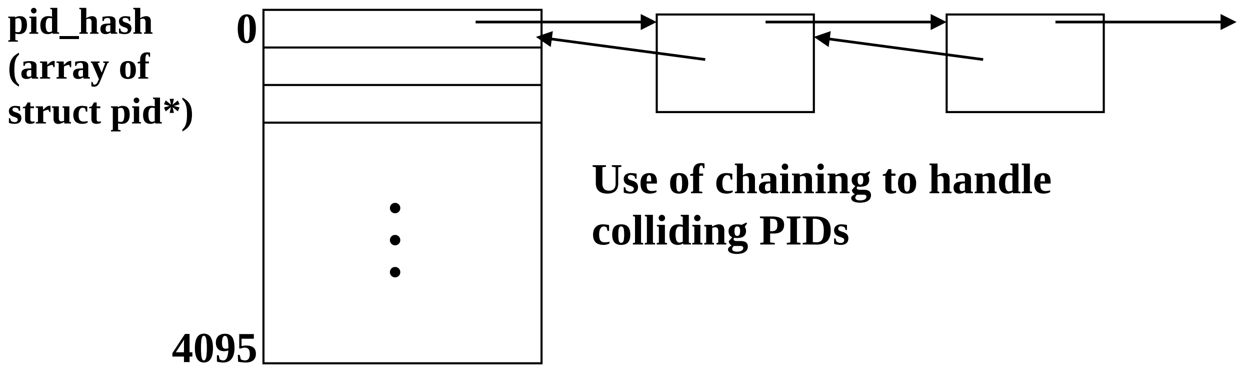
Task Identification and Linkage (cont.)

- Task structures
  - placed in a cyclic, doubly-linked list
  - list starts with sentinel: `init_task`
    - first task created by kernel at boot time
    - persists until machine shut down/rebooted



Task Identification and Linkage (cont.)

- For system calls
  - must translate pid to struct pid* (small structure referencing task)
  - **`find_pid`** (kernel/pid.c) uses a hash table
    - map large, sparse space into small, dense space
  - doubly-linked list per bin, but back links only for deletion
  - hash function = ((pid * big prime #) >> shift)



Creating Processes/Tasks

- User-level: fork, vfork, and clone system calls
- In kernel: all map into **`do_fork`**

```
int do_fork (unsigned long clone_flags,
             unsigned long stack_start,
             struct pt_regs* regs,
             unsigned long stack_size,
             int __user* parent_tidptr,
             int __user* child_tidptr);
```

  - **`__user`** means not to be dereferenced
  - returns new pid or negative value on error

Creating Processes/Tasks (cont.)

- Parameters
  - clone_flags – control flags, e.g., CLONE_PARENT, which creates a sibling task (like a thread) instead of a child task
  - stack_start – the new task's ESP (in user space; 0 for kernel threads)
  - regs – register values for the new task
  - stack_size – ignored on x86, but needed on some ISAs
  - parent/child_tidptr – filled in as part of system call (**`sys_clone`** only)

Creating Processes/Tasks (cont.)

- **`do_fork`** calls **`copy_process`** (same file) to set up the process descriptor and any other kernel data structures necessary for child execution (e.g., thread_info structure and Kernel Mode stack)
  - only stack_start and regs are used by x86 version

Copy on Write

- How do you start a program?
  - at the user level, some other program has to start it
  - usually started by a shell or other interface program
- The mechanism used consists of two parts
  - fork, which creates a copy of the current program
  - exec, which loads a new program and starts it

Copy on Write

- However, the common case is fork quickly followed by exec

- fork duplicates the process’ address space
  - painfully slow in early systems
  - especially when data were then mostly discarded (on exec)

- BSD added vfork (virtual fork)
  - parent blocks while child uses address space
  - after child exits, control of address space returns to parent

Copy on Write

- Mach added copy-on-write
  - instead of duplicating data
    - duplicate page tables
    - turn off write permission
  - when either process tries to write to a given page, give it a private copy

- Copy on write is an example of a “lazy” approach to work (opposite is “eager”)
  - allows us to avoid work that won’t actually be useful
  - we’ll see more lazy approaches in the future

Creating Kernel Threads

- What is a kernel thread?
  - a task without an associated address space
  - inherits address space from last user task to execute
  - (all address spaces map the kernel pages and data structures)

- The interface
  - prototype in asm/processor.h; implementation in arch/i386/kernel/process.c

```
int kernel_thread (int (*fn)(void*), void* arg,
                   unsigned long flags);
```

  - returns new pid or negative value on error

Creating Kernel Threads (cont.)

- init_task is a kernel thread
  - created during boot (in start_kernel); persists until shutdown
  - pid = 1

init task (pid=1)

init_mm (all kernel pages are mapped)
init_fs
init_files
init_signals/init_sighand

- Other kernel threads
  - ksoftirqd – the soft interrupt daemon (periodically try to execute tasklets)

x86 Task State Segment

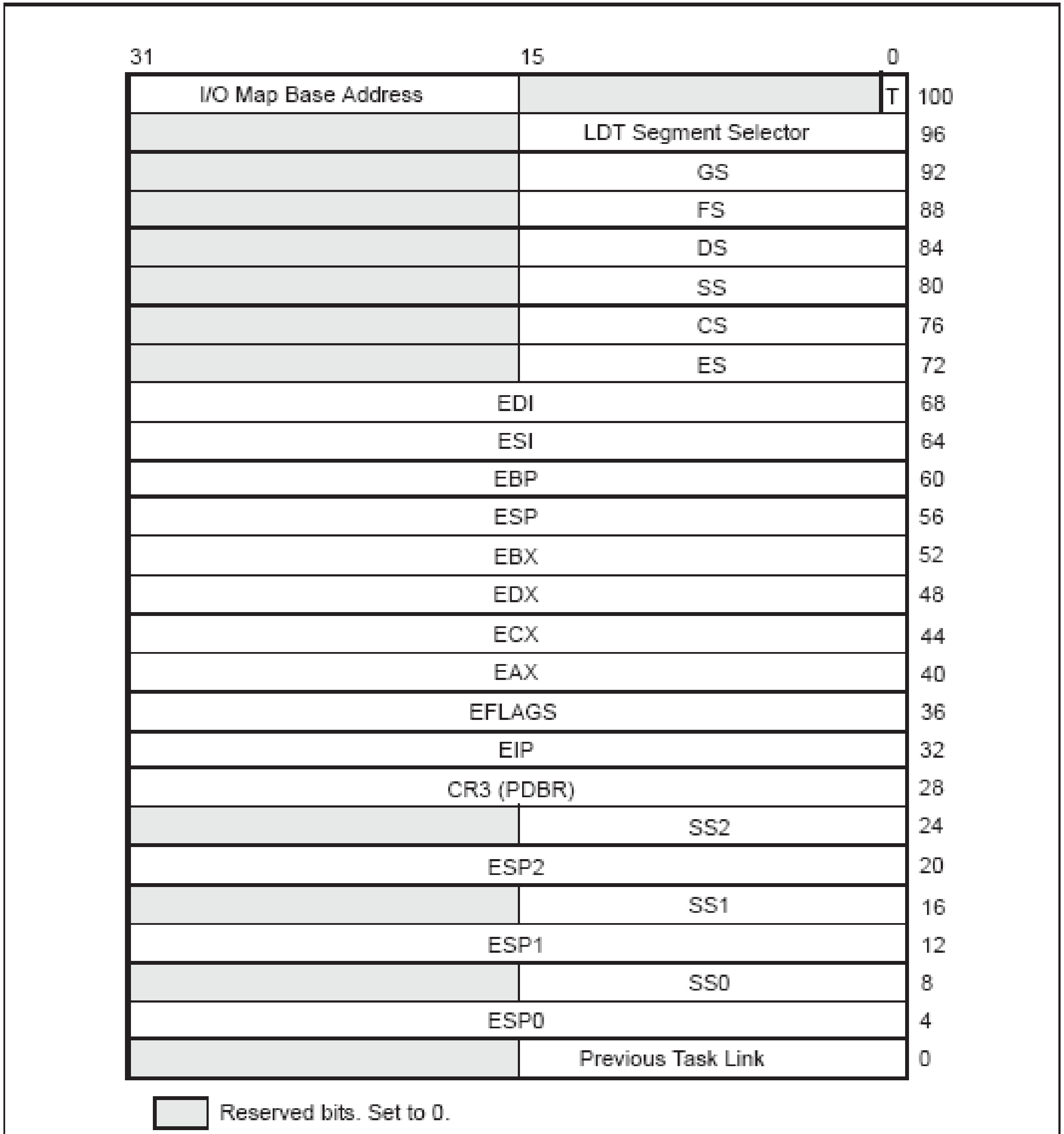


Figure 6-2. 32-Bit Task-State Segment (TSS)

Task State Segment Transparency

- TSS contains three parts
  - first part is shown in diagram
  - second two parts are variable-length
    - interrupt redirection (for emulation of earlier ISA generations)
    - I/O permission bitmap (for individual ports)
  - I/O Map Base Address gives pointer to start of the last part
  - Segment limit (in TSS descriptor) defines I/O bitmap length
- T field – debug trap flag
  - if set, execution stops (exception thrown) when task executes

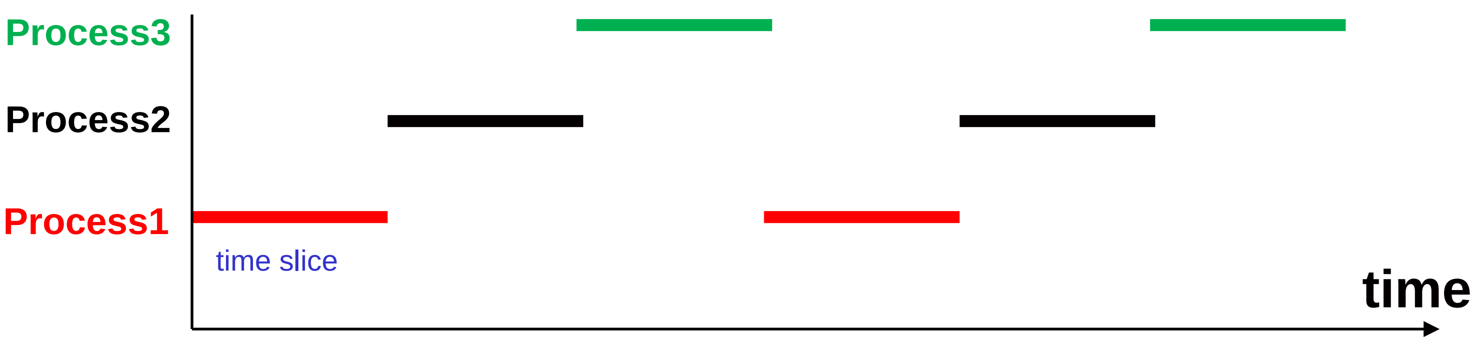
Task State Segment Transparency (cont.)

- SS/ESP fields are saved values for user privilege level
  - Other privilege levels read from SS2/ESP2, SS1/ESP1, SS0/ESP0
- I/O bitmap privilege checking
  - IOPL (I/O Privilege Level field in EFLAGS register) specifies level needed for arbitrary access
  - if CPL > IOPL
    - processor checks I/O bitmap in TSS
    - exception generated unless
      - bitmap contains enough bits to represent port as a bit
      - bit representing port (I/O port for *in/out* instruction) is set to 0
  - Linux keeps only the first 1024 ports in bitmap to conserve space

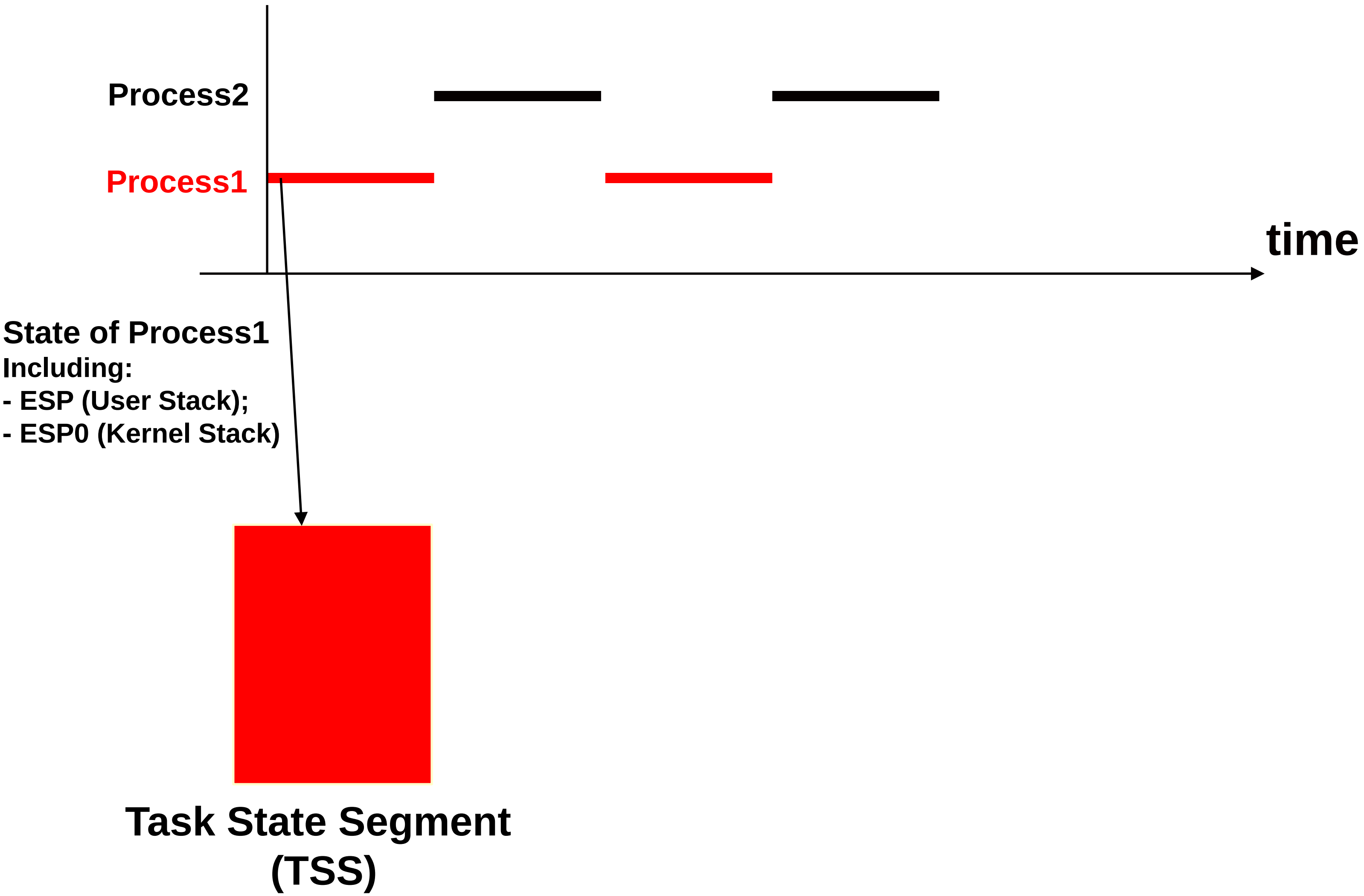
Process/Task Switch

process1 -> process2 -> process3

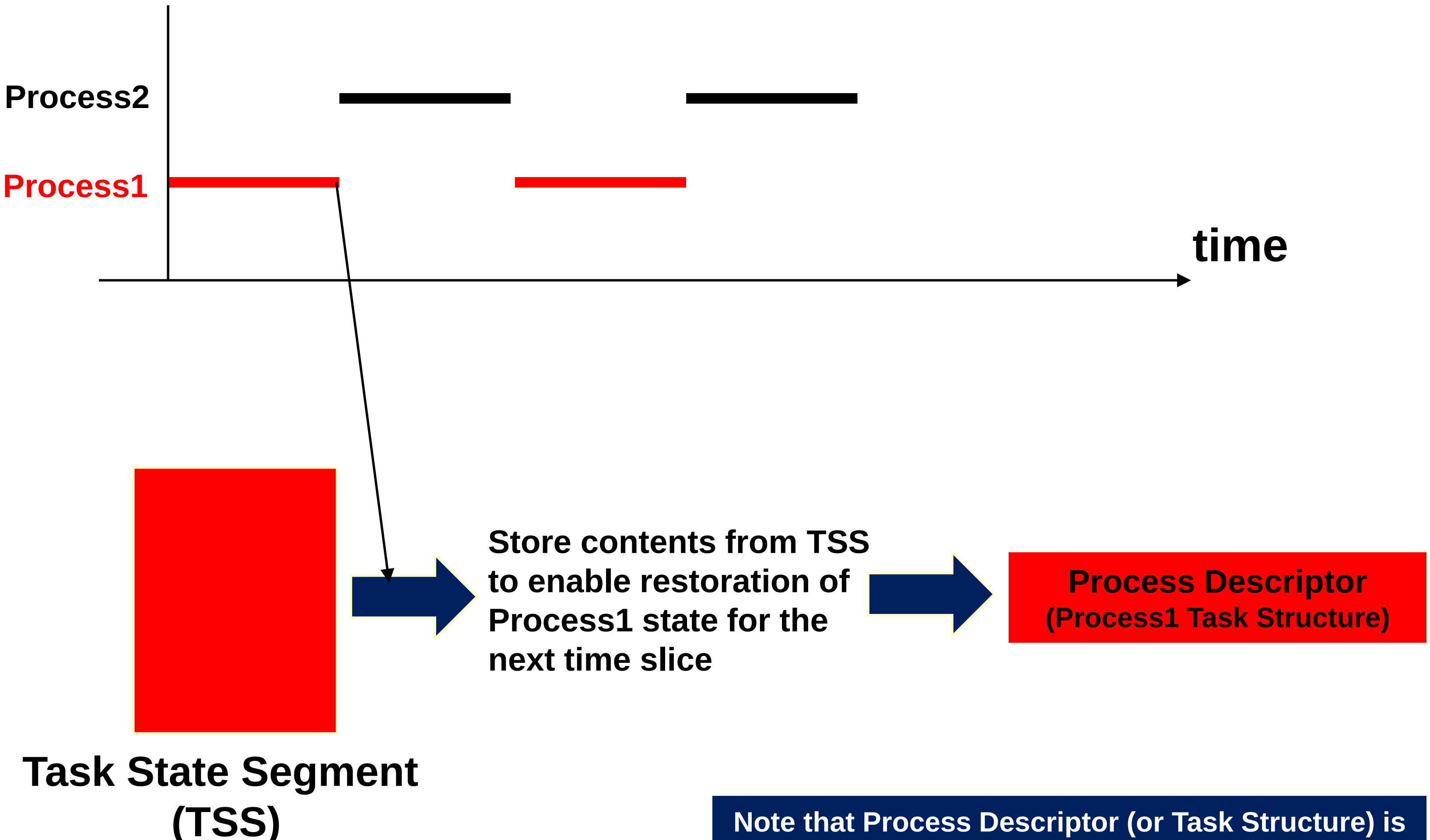
Round Robin



Use of TSS in Task Switching

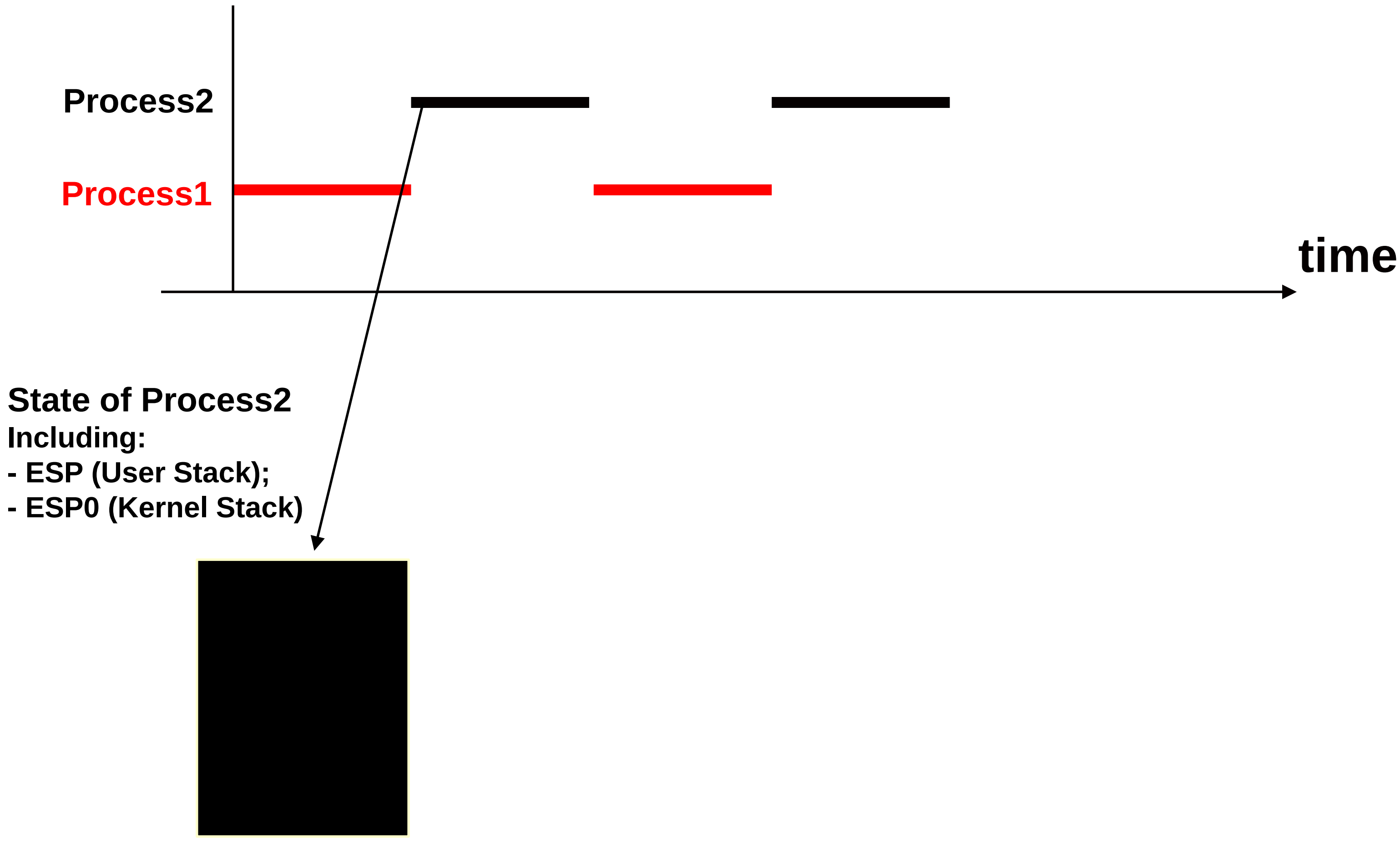


Use of TSS in Task Switching



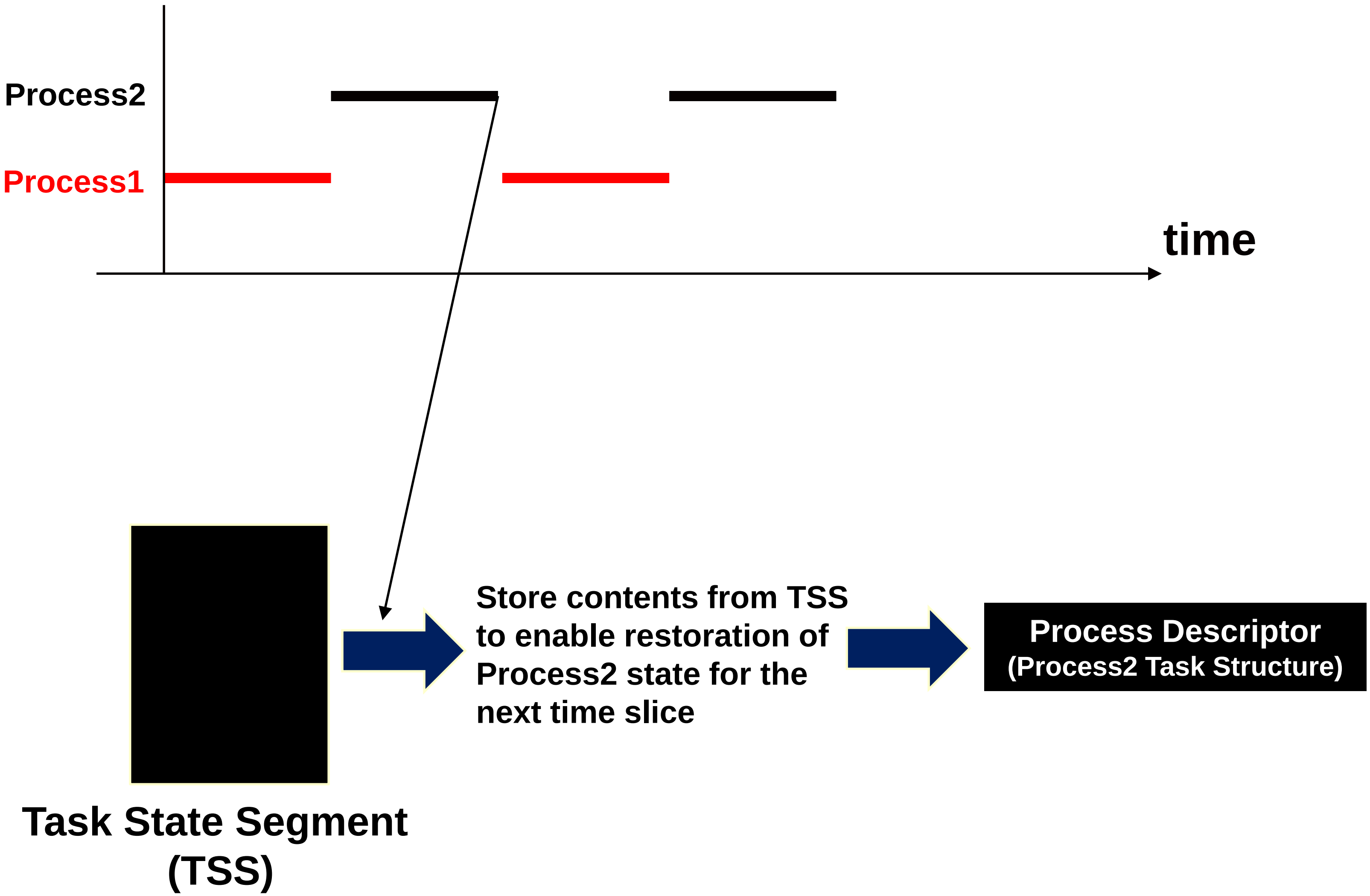
Note that Process Descriptor (or Task Structure) is equivalent to PCB (Process Control Block in MP3)

Use of TSS in Task Switching

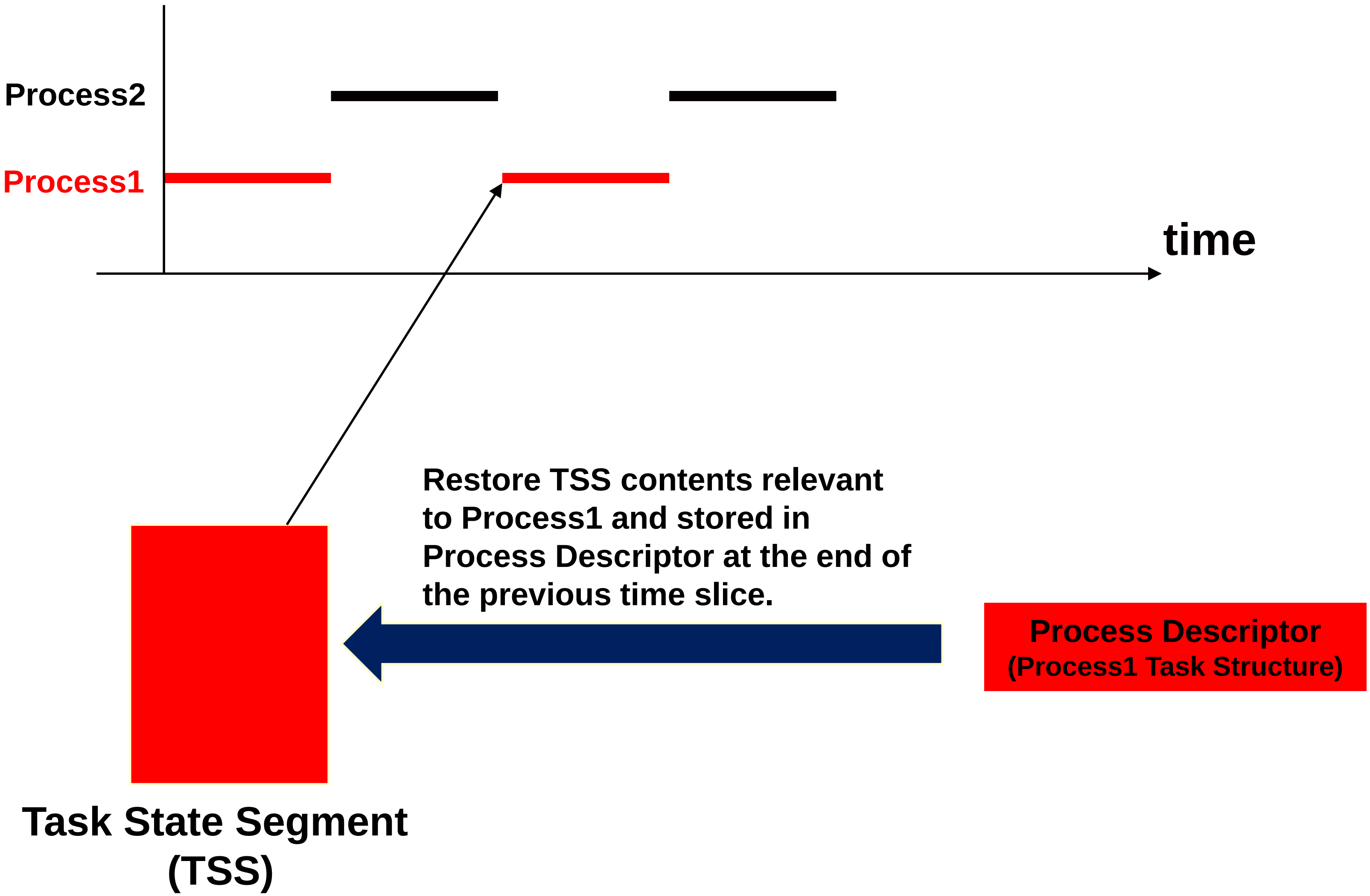


Task State Segment (TSS)

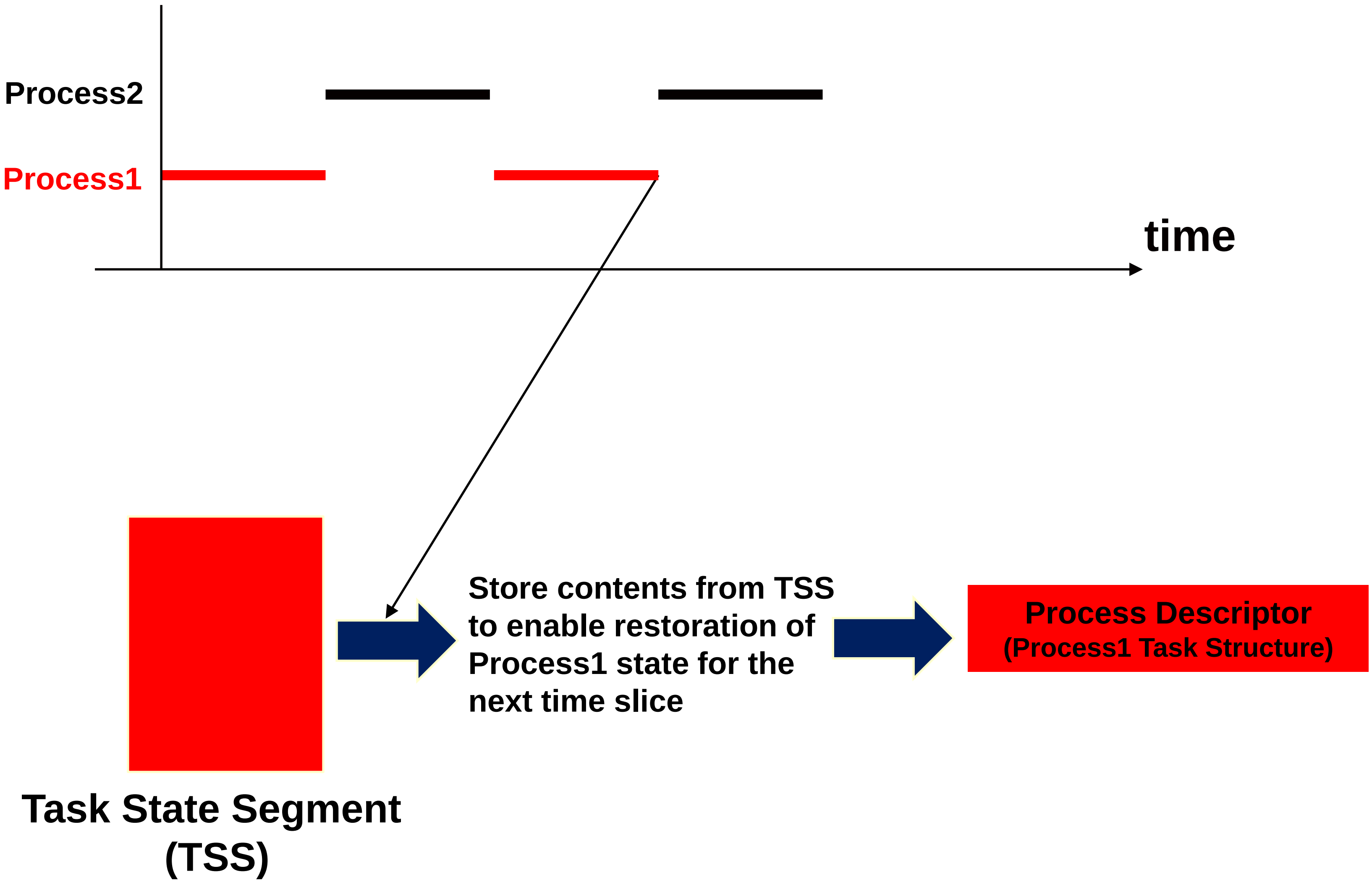
Use of TSS in Task Switching



Use of TSS in Task Switching



Use of TSS in Task Switching



Use of TSS in Task Switching