STEVENS
INSTITUTE *of* TECHNOLOGY
THE INNOVATION UNIVERSITY®

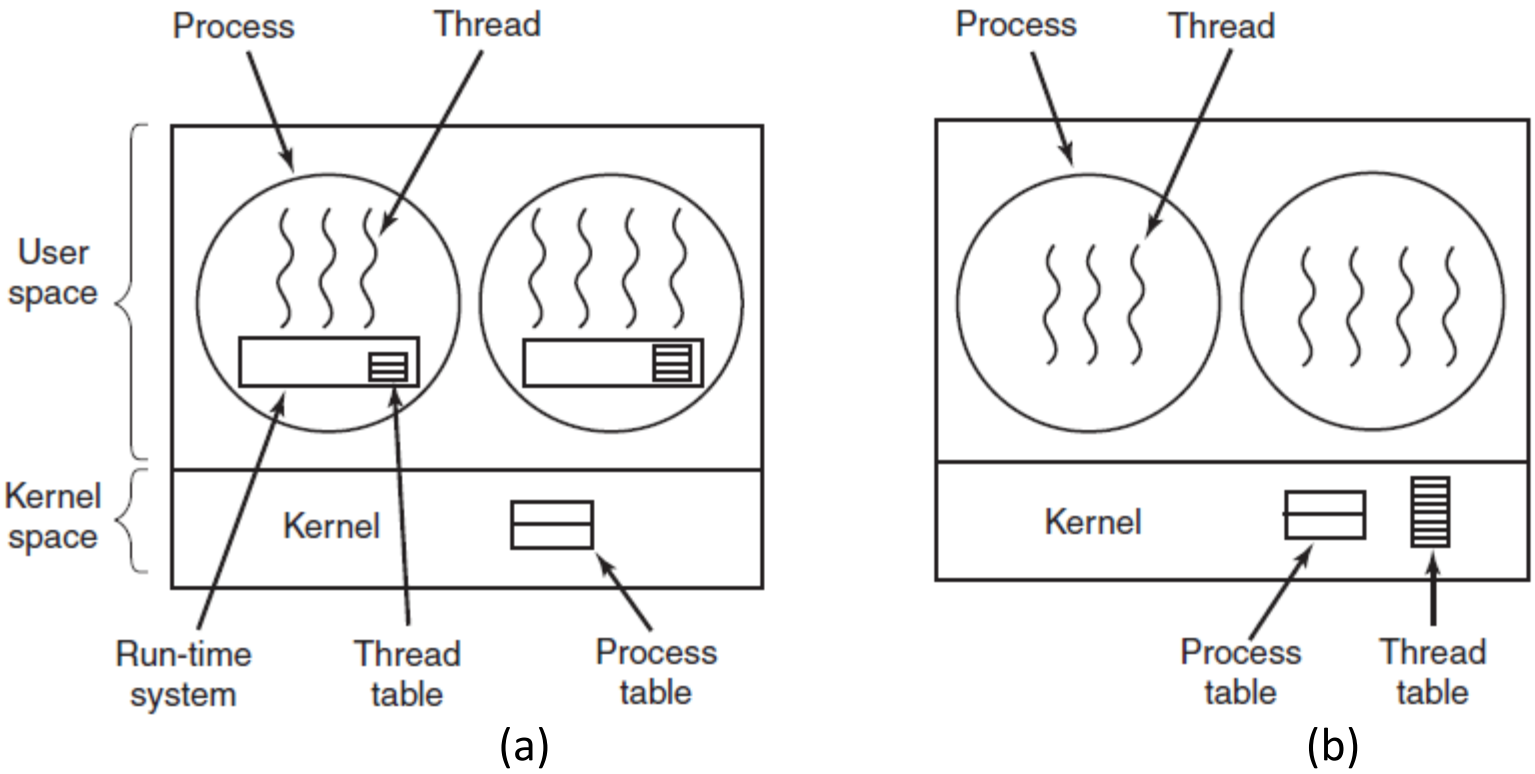
CS 492: Operating Systems

Threads (2)

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

Implementing Threads in User Space



(a) A user-level threads package.
(b) A threads package managed by the kernel.

Pros of Threads in User Space

- A user-level threads package can be implemented on an OS that does not support threads
- Thread switching is at least an order of magnitude faster, than trapping to the kernel
- Thread scheduling very fast: no context switching, no kernel trap, no flushing of memory cache
- Each process can have its own scheduling algorithm

Cons of Threads in User Space

- Blocking System calls (eg. waiting for keyboard input)
- Page faults (partial load of programs into memory)
- Threads need to voluntarily give up the CPU for multiprogramming
- Programmers generally want threads precisely in applications where threads block often (Web Server)

Pros of Threads in Kernel

- No run-time system needed in each process
- No thread table in each process
- Blocking system calls are not a problem, since the kernel scheduler can schedule another thread in that case

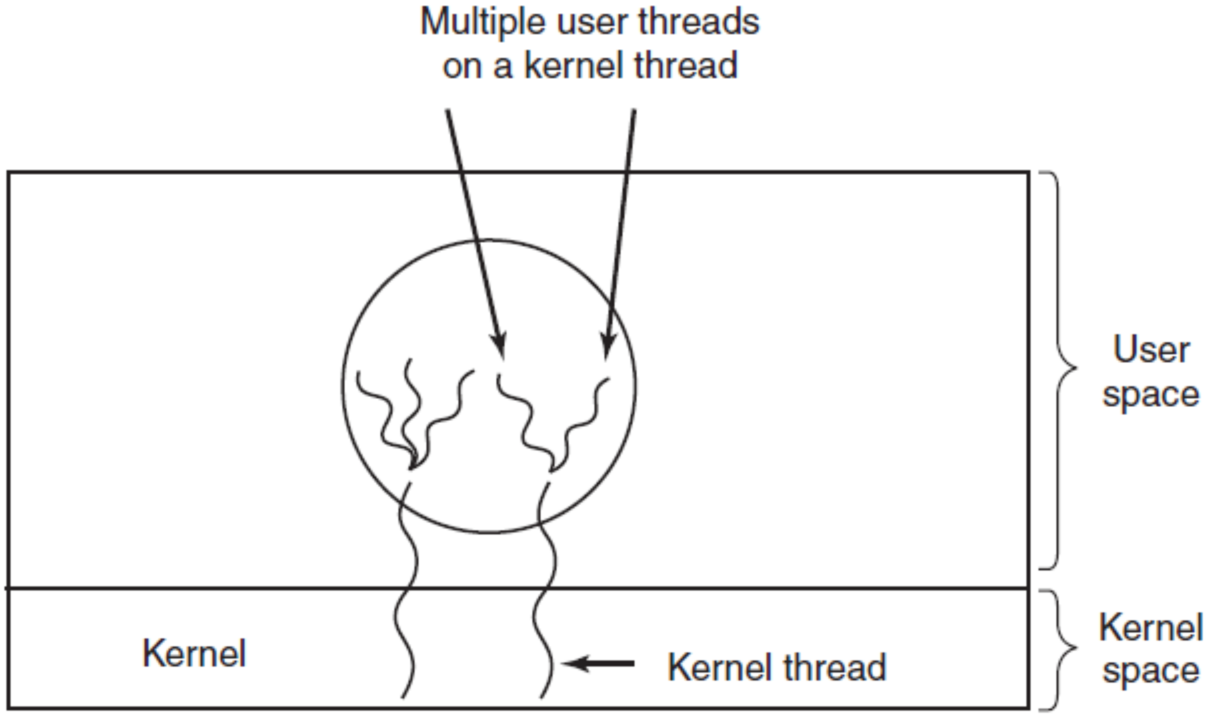
Cons of Threads in Kernel

- If thread operations are common (creation, termination), much more kernel overhead will be incurred
- Fork a multithreaded process?
- Signals sent to processes. Should the kernel assign it to a specific thread to handle?
- Slower than user-space threads

Question?

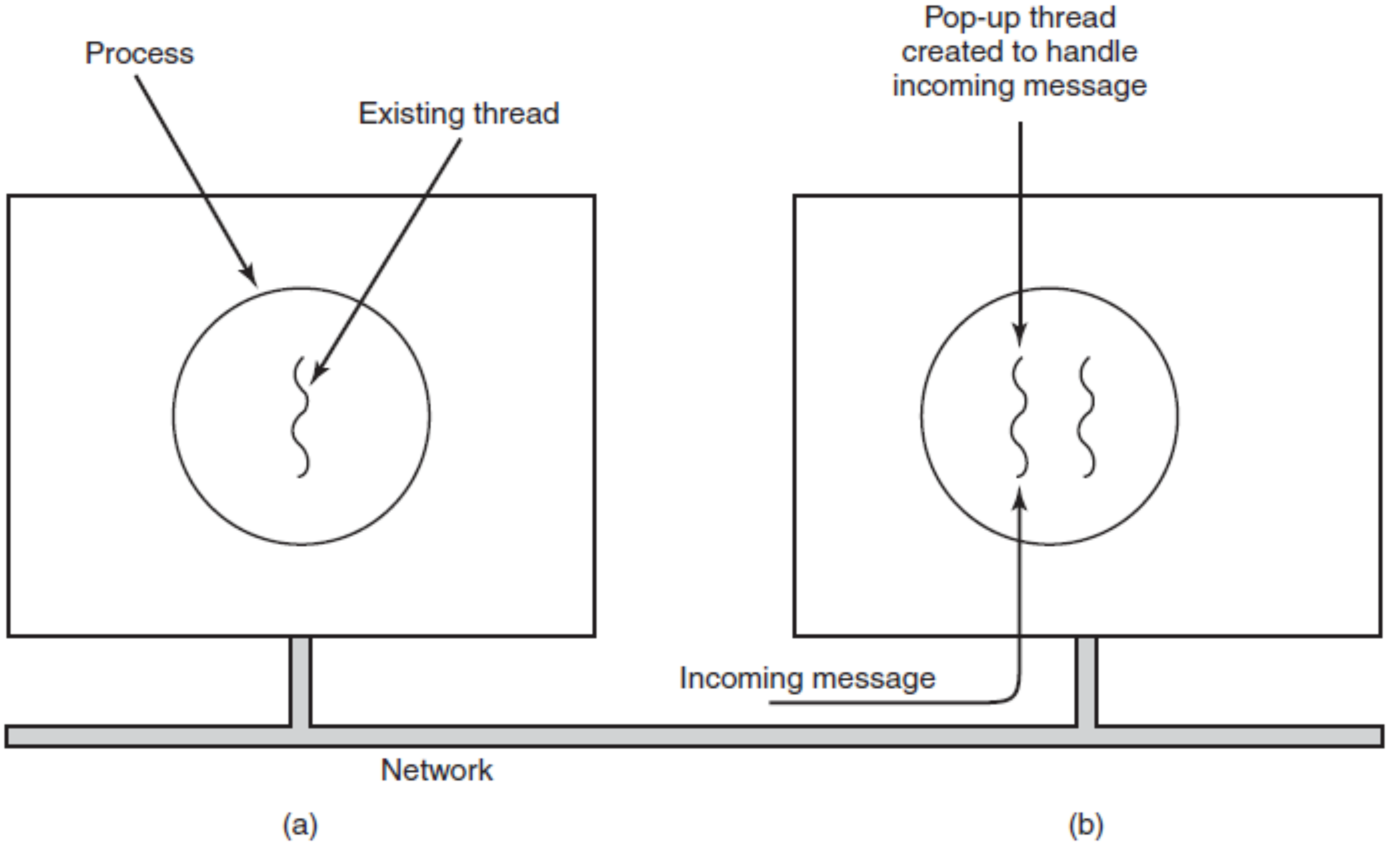
- What is the biggest advantage of implementing threads in user space?
- What is the biggest disadvantage?

Hybrid Implementations



Multiplexing user-level threads
onto kernel-level threads.

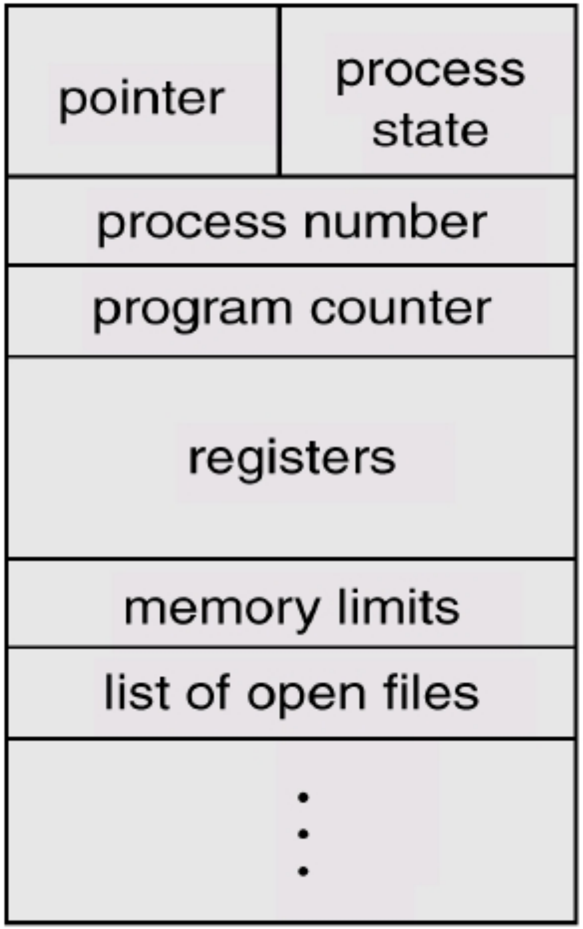
Pop-up Threads



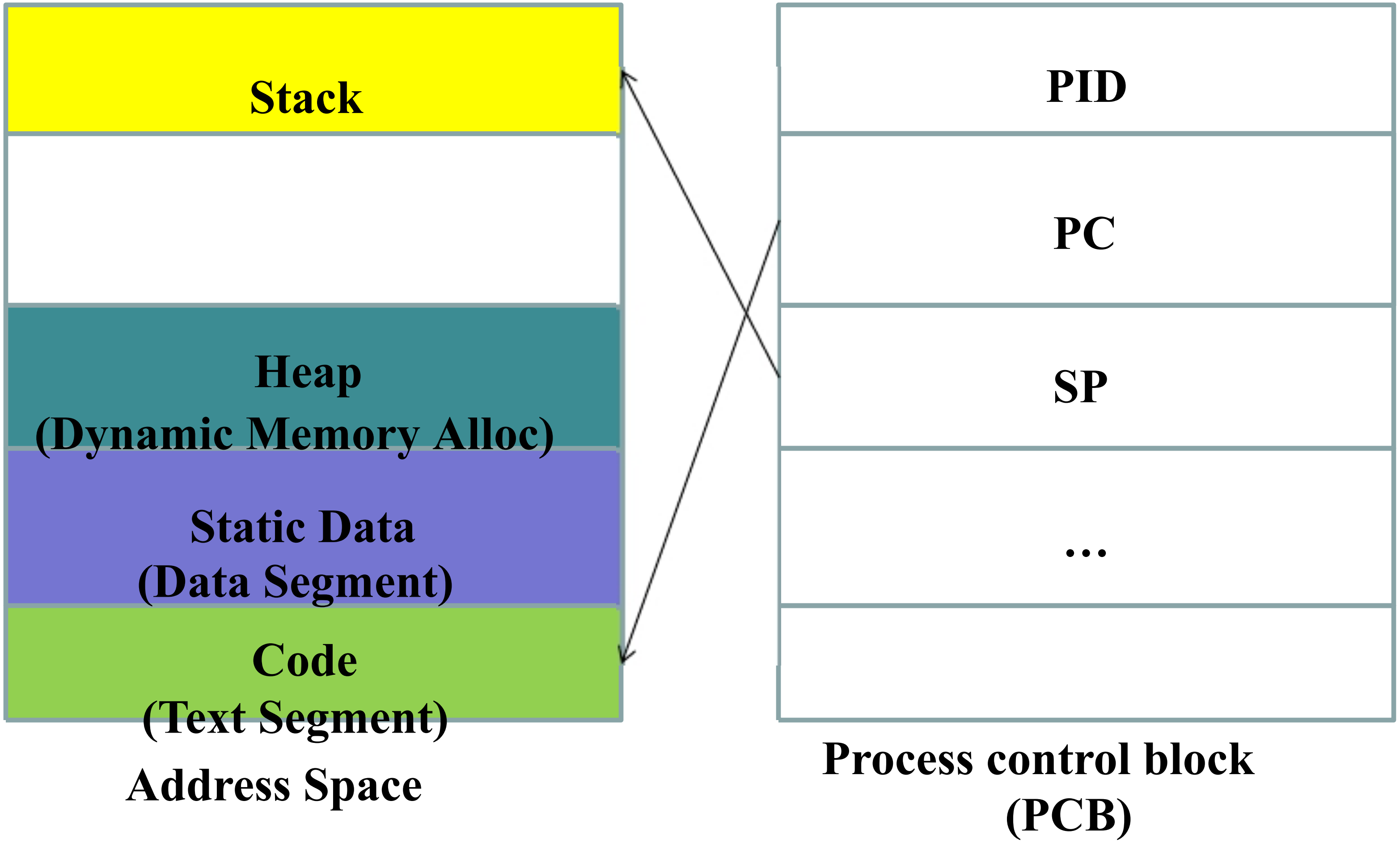
Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

Recall: Process Control Block (PCB)

- Each PCB contains
  - Process state
  - Process ID
  - Program Counter (PC)
  - Current CPU registers (if not executing)
  - CPU scheduling info (e.g., priority)
  - Memory-management info
  - Resources allocated to it
  - Resources it needs (e.g., I/O status information)

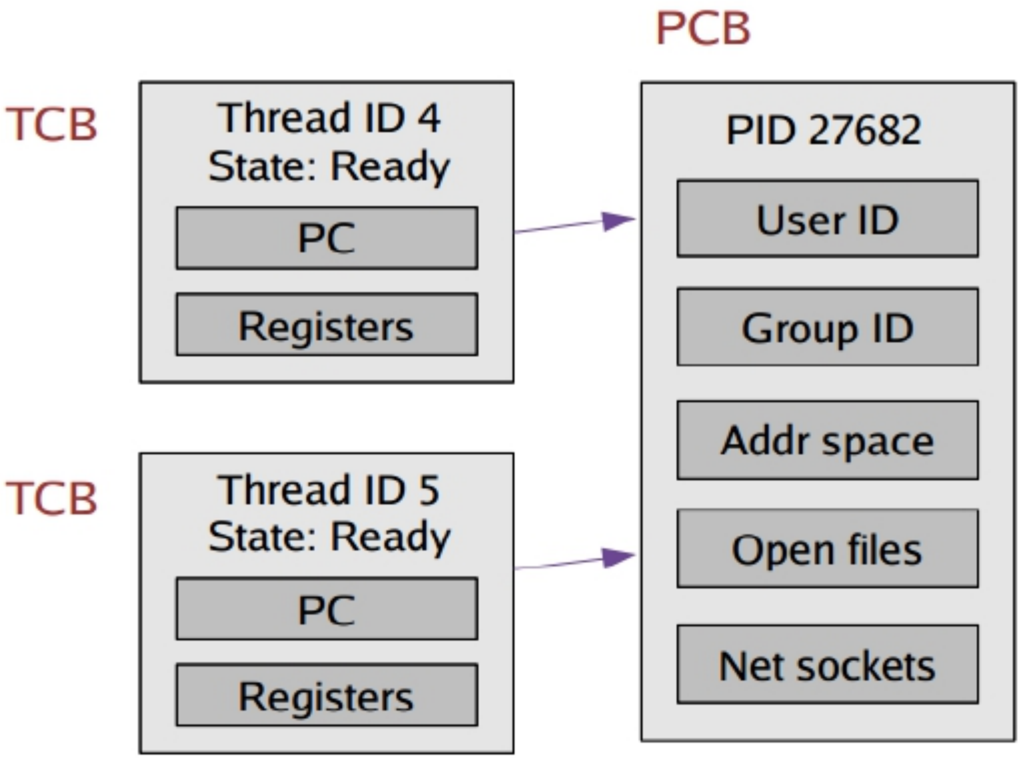


PCB Diagram

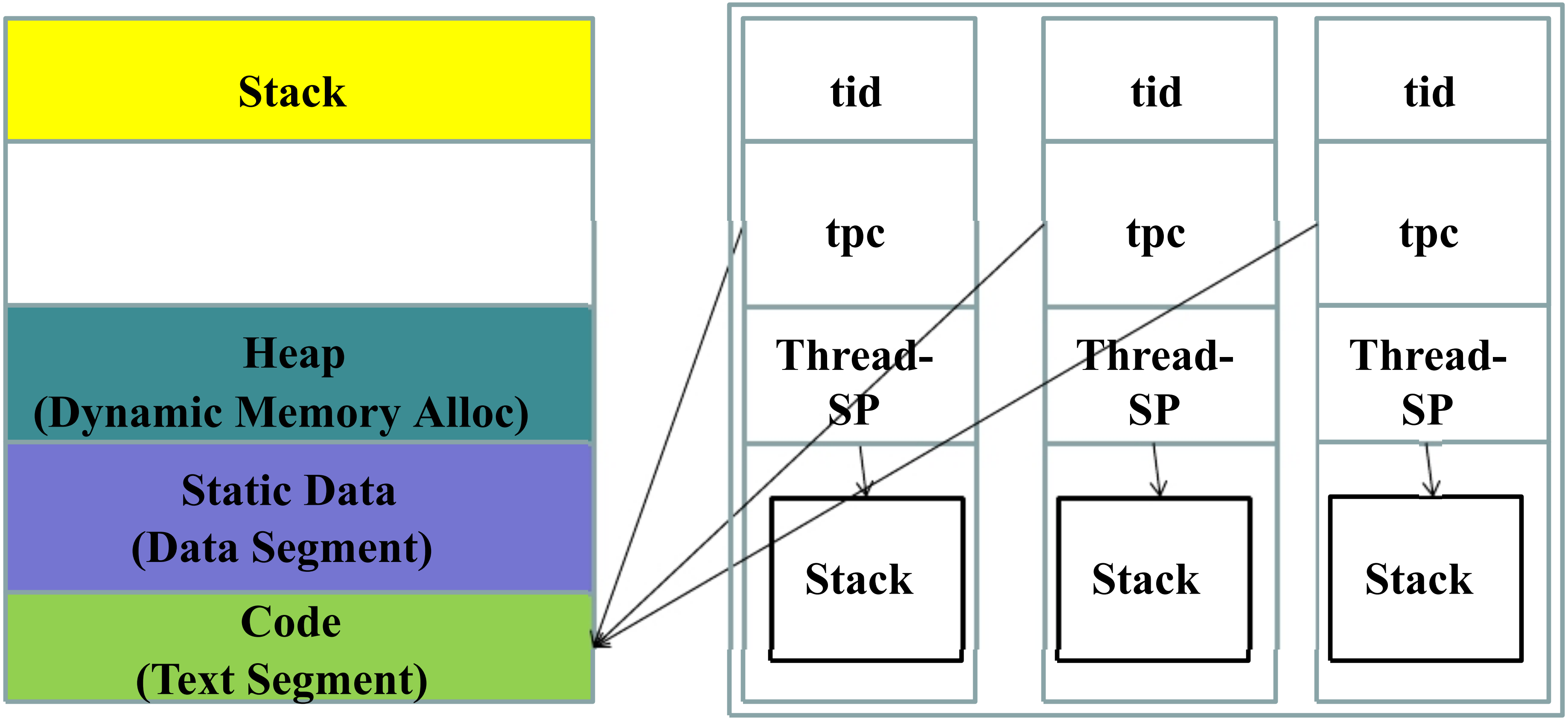

Stack
Heap
(Dynamic Memory Alloc)
Static Data
(Data Segment)
Code
(Text Segment)
Address Space
PID
PC
SP
…
Process control block
(PCB)

Thread Control Blocks (TCBs)

- Idea: Break the PCB into two pieces:
  - Thread-specific stuff: Process state
  - Process-specific stuff: Address space and OS resources (open files, etc.)



TCB Diagram



Address Space

Thread Control Block (TCB)

Thread Control Block (TCB)

- Information directly related to process execution: stored in Thread Control Block (TCB)
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Pending I/O information
- Other information associated with processes: stored in Process Control Block (PCB)
  - Memory management information
  - Accounting information

Thread Control Block (TCB)

- TCB's are smaller and cheaper than processes
  - Linux TCB (thread_struct) has 24 fields
  - Linux PCB (task_struct) has 106 fields

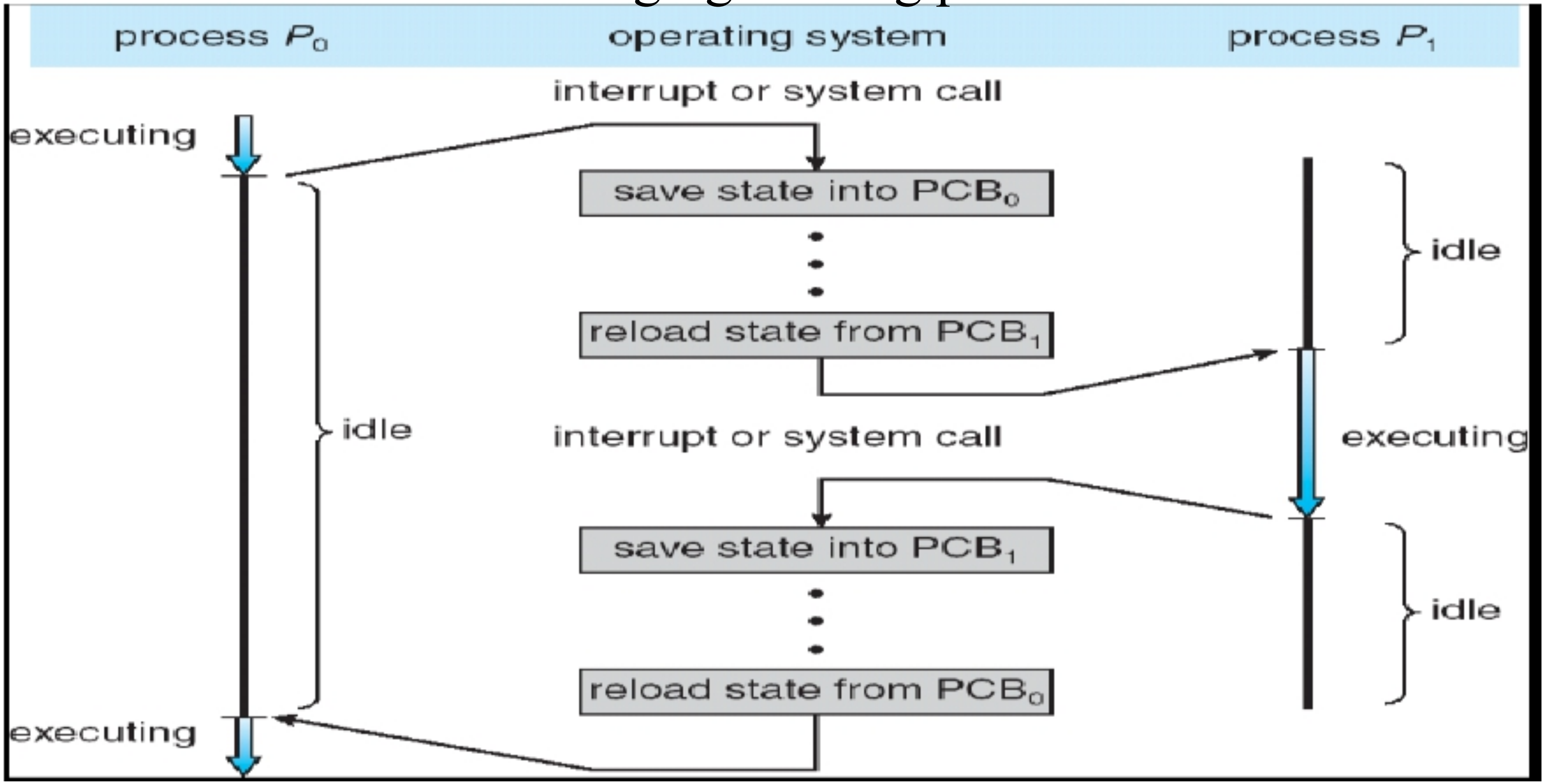
Effect of TCBs

- Threads in a process can execute different parts of the program code at the same time.

- Threads can execute the same parts of the code at the same time, but with different execution state:
  - They have independent current instructions.
  - They are working with different data.

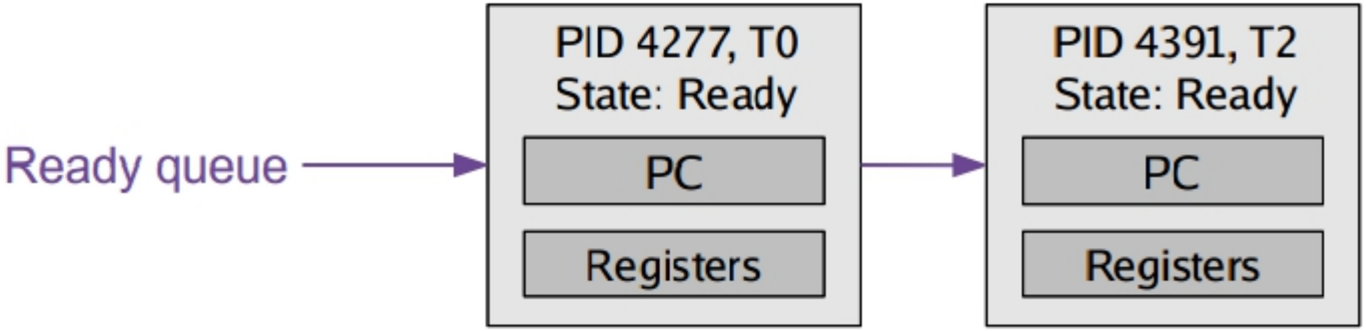
Recall: Context Switch

- *Context switch: c*hanging running proc



Now Context Switching with TCB

- TCB is now the unit of a context switch
  - – Ready queue, wait queues, etc. now contain pointers to TCB's
  - – Context switch causes CPU state to be copied to/from the TCB



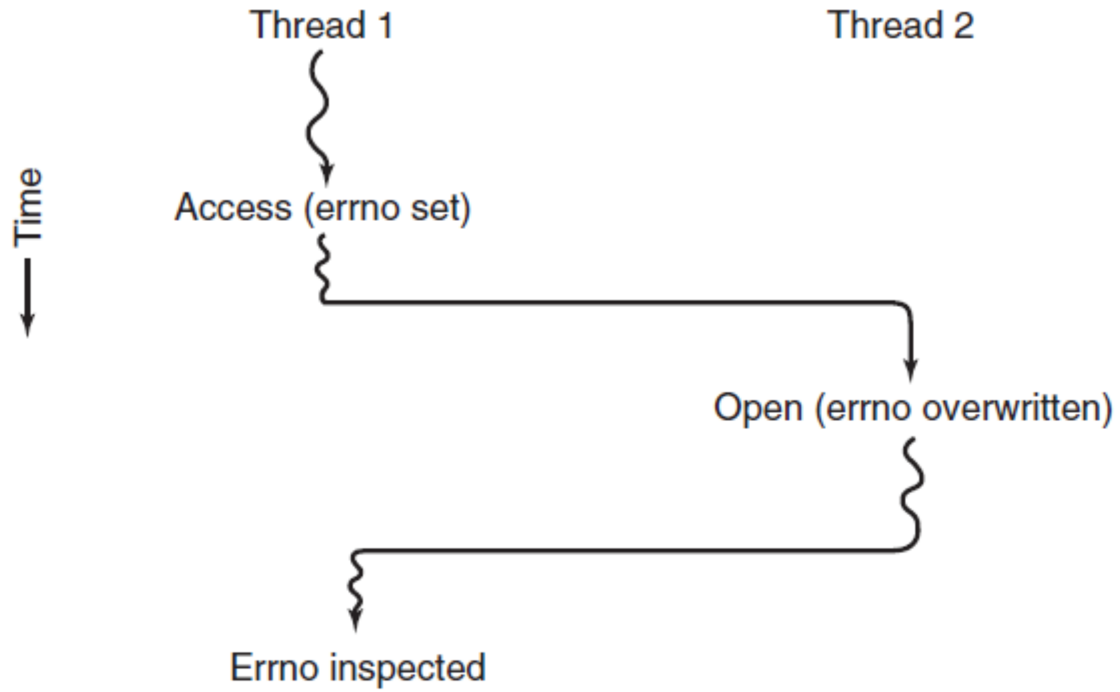
Context Switching (Cont.)

- Context switch between two threads in the same process:
  - No need to change address space
- Context switch between two threads in different processes:
  - Must change address space, sometimes invalidating cache
  - This will become relevant when we talk about virtual memory.

Concept Revisit

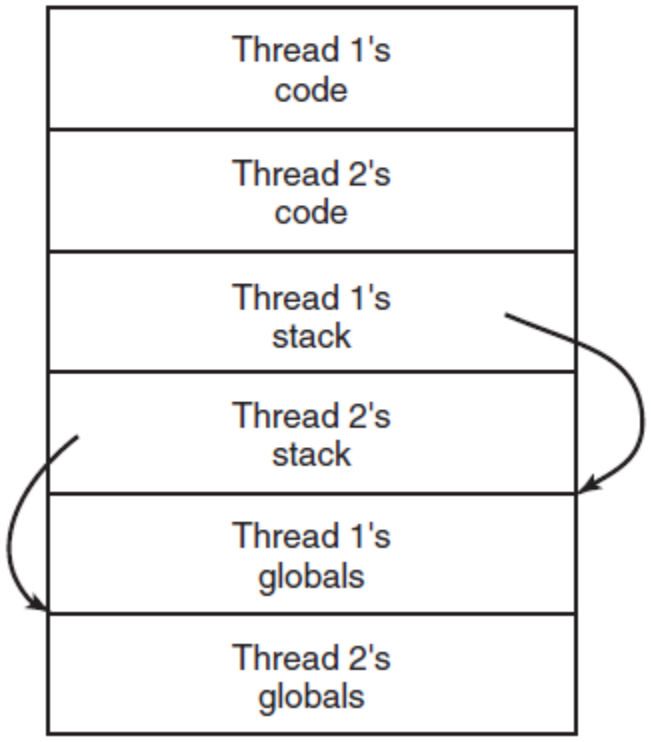
- Thread = an **independent** sequential execution stream within process
- The independency is accomplished because a thread maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties (such as policy or priority)

Making Single-Threaded Code Multi-Threaded



Conflicts between threads over the use of a global variable.

Making Single-Threaded Code Multi-Threaded (2)



Threads can have private global variables.

Making Single-Threaded Code Multi-Threaded (3)

- create_global(“bufptr”); //allocates storage for a pointer
- set_global(“bufptr”, &buf);
- bufptr = read_global(“bufptr”);
  - We used a procedure create global to allocate storage for a pointer to the variable, rather than the variable itself. Is this essential, or could the procedures work with the values themselves just as well?

Making Single-Threaded Code Multi-Threaded (4)

- Library procedures not reentrant
- Memory allocation procedures
- Signals (especially in user-level threads)
- Stack management