CSC 256/456: Operating Systems

Threads and Interprocess Communication

John Criswell
University of Rochester

Today

- Inter-process Communication (IPC)
  - Shared Memory
  - Message Passing
- Threads
  - Thread Concept
  - Multi-threading Models
  - Types of Threads

But first, a note about commenting…

Comments Not for Judging Future Self

XKCD Comic #1421
By Randall Munroe
xkcd.com

Function Comments

```
/*
 * Intrinsic: sva_swap_integer()
 *
 * Description:
 *  This intrinsic saves the current integer state and swaps in a new one.
 *
 * Inputs:
 *  newint - The new integer state to load on to the processor.
 *  statep - A pointer to a memory location in which to store the ID of the
 *           state that this invocation of sva_swap_integer() will save.
 *
 * Return value:
 *  0 - State swapping failed.
 *  1 - State swapping succeeded.
 */
uintptr_t
sva_swap_integer (uintptr_t newint, uintptr_t * statep) {
  /* Old interrupt flags */
  uintptr_t rflags = sva_enter_critical();

  /* Pointer to the current CPU State */
  struct CPUState * cpup = getCPUState();

  /*
   * Get a pointer to the memory buffer into which the integer state should be
   * stored.  There is one such buffer for every SVA thread.
   */
  struct SVAThread * oldThread = cpup->currentThread;
```

What about the programming assignment handin procedure?

Still Working on It

- Handin process will be announced on Blackboard by Thursday lecture
- Programming Assignment #1
  - Due ***Thursday at 11:59 pm!***

Where's the PCB?!

It Depends!

FreeBSD

Linux

Scheduling: Transferring Context Blocks

Coroutines

transfer(other)

<u>push callee-saved registers</u>

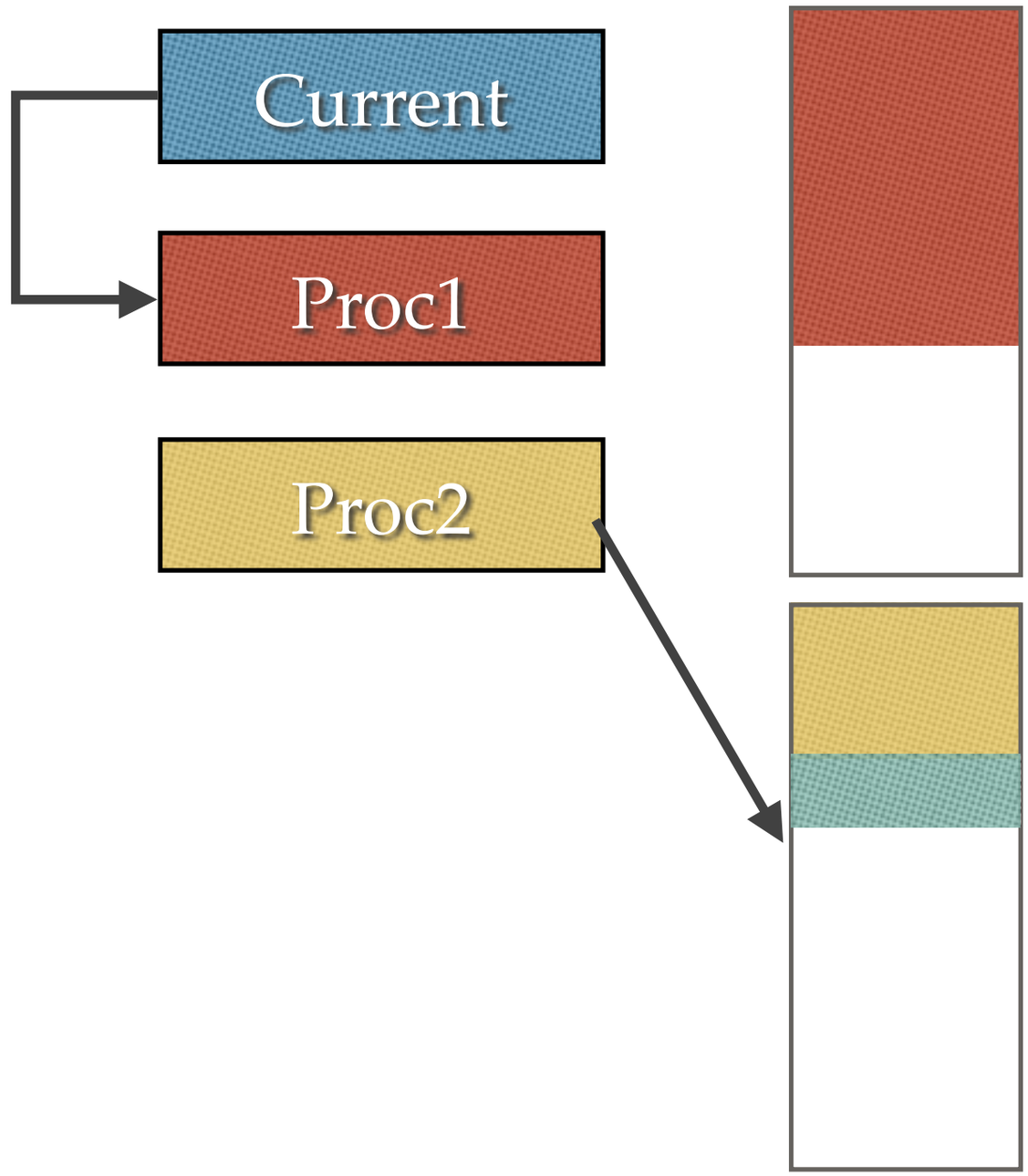
*current := sp

current := other

sp := *current

<u>pop callee-saved registers (sans sp)</u>

return (into different coroutine!)



Scheduling: Transferring Context Blocks

Coroutines

transfer(other)

<u>push callee-saved registers</u>

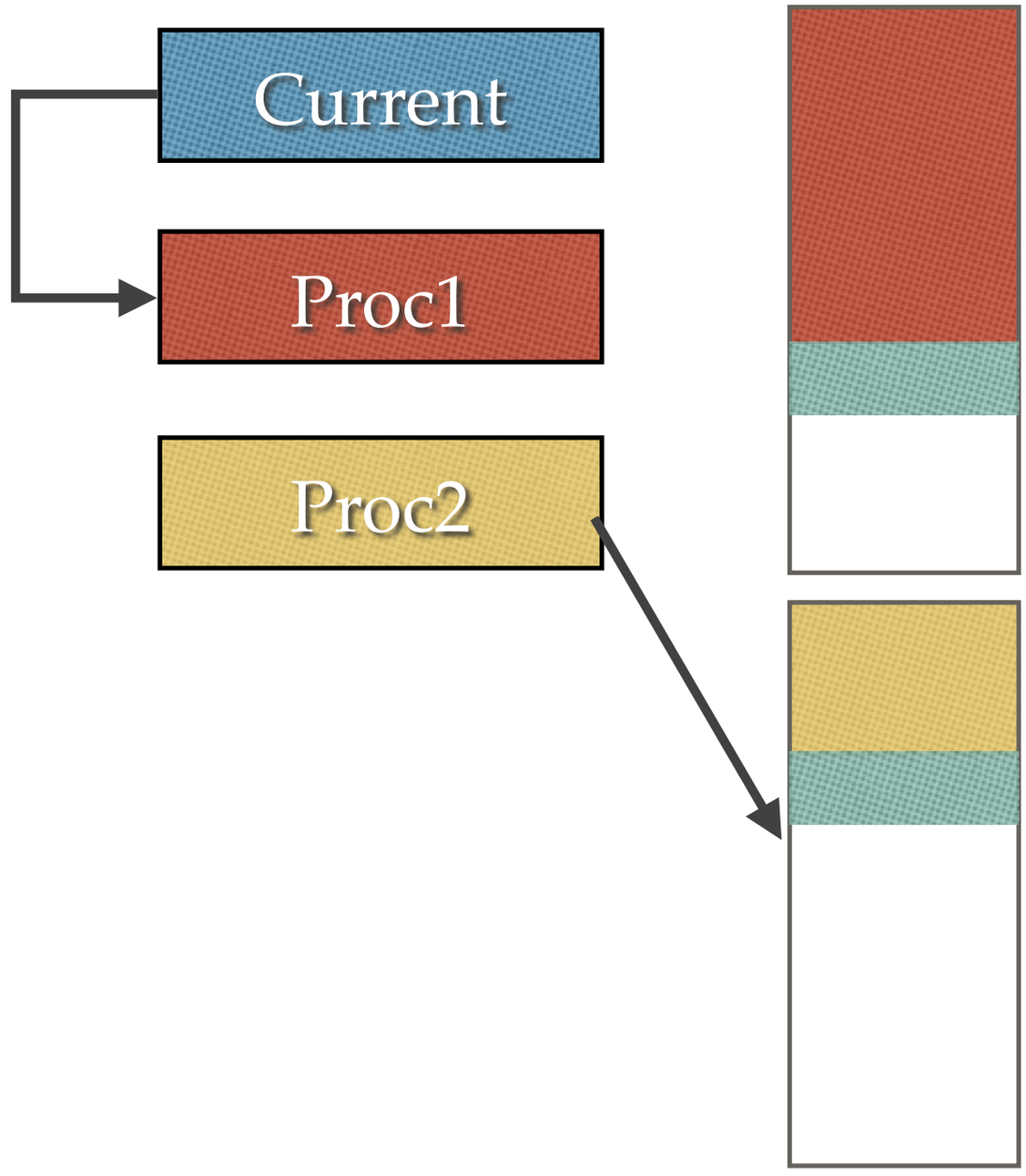
*current := sp

current := other

sp := *current

<u>pop callee-saved registers (sans sp)</u>

return (into different coroutine!)



Scheduling: Transferring Context Blocks

Coroutines

transfer(other)

<u>push callee-saved registers</u>

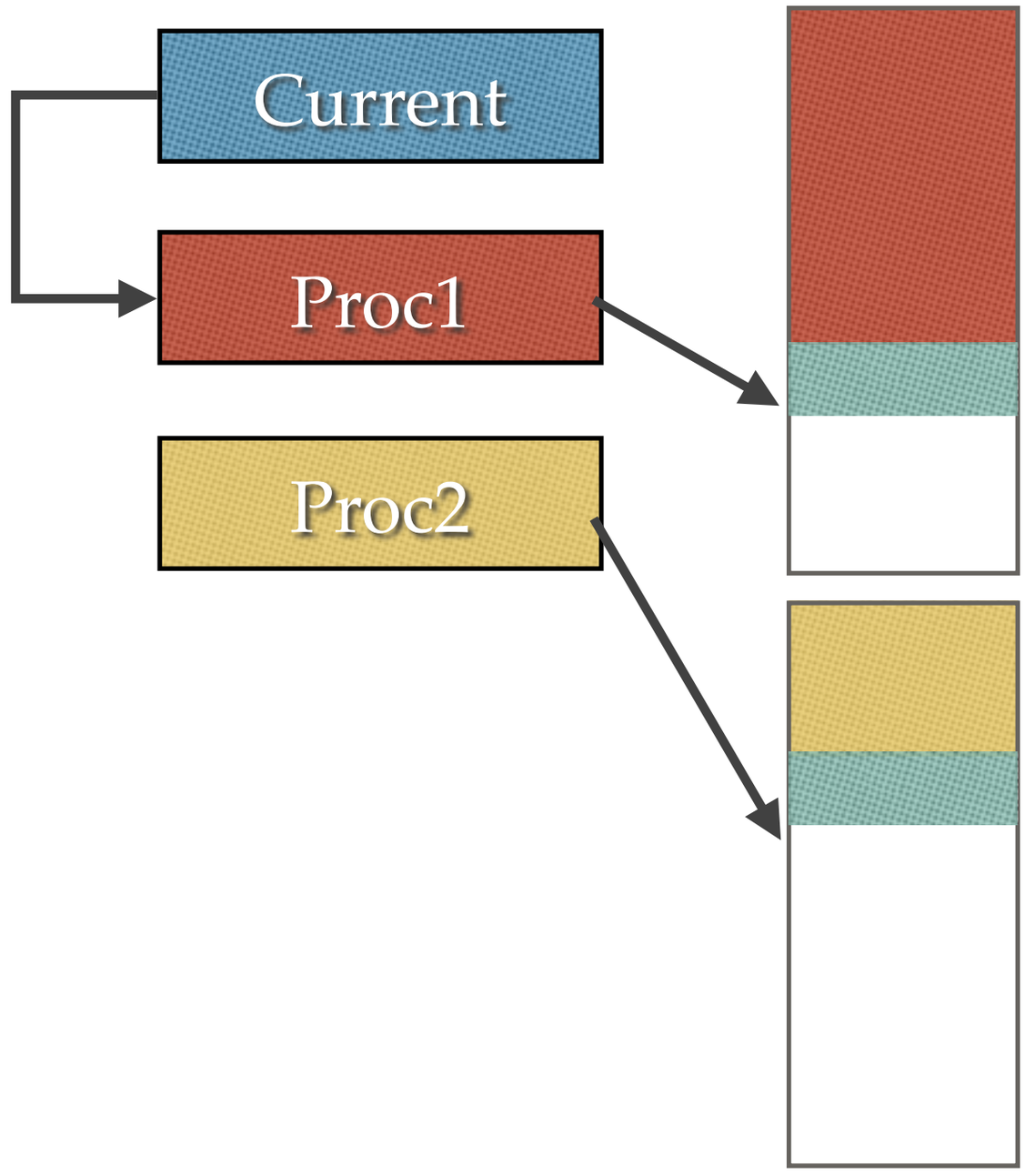
*current := sp

current := other

sp := *current

<u>pop callee-saved registers (sans sp)</u>

return (into different coroutine!)



Scheduling: Transferring Context Blocks

Coroutines

transfer(other)

<u>push callee-saved registers</u>

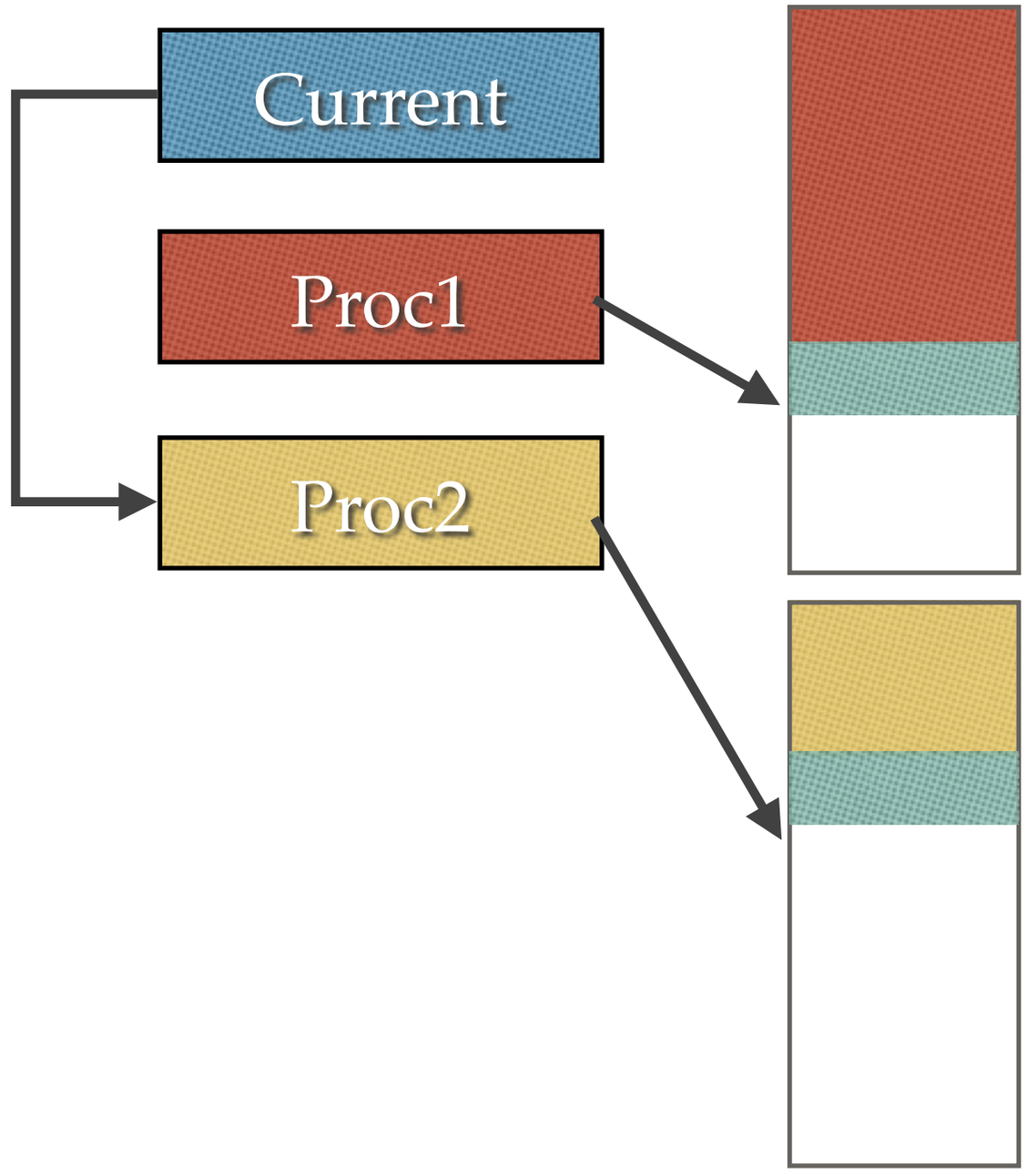
*current := sp

current := other

sp := *current

<u>pop callee-saved registers (sans sp)</u>

return (into different coroutine!)



Scheduling: Transferring Context Blocks

Coroutines

transfer(other)

<u>push callee-saved registers</u>

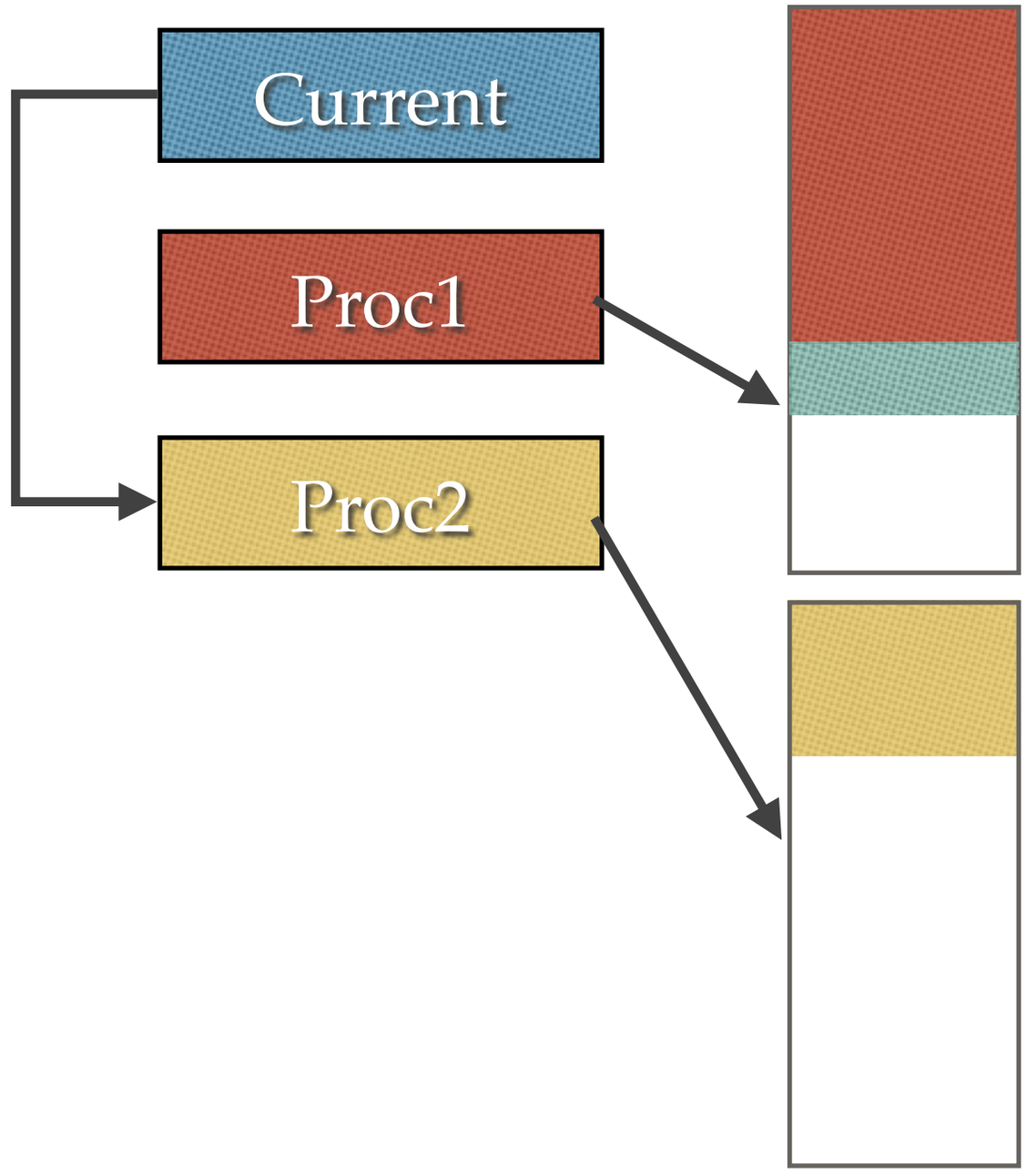
*current := sp

current := other

sp := *current

<u>pop callee-saved registers (sans sp)</u>

return (into different coroutine!)



Interprocess Communication

Interprocess Communication

- Reasons for processes to cooperate
  - Information sharing (e.g., files)
  - Computation speedup
  - Modularity and protection
  - Convenience - multitasking

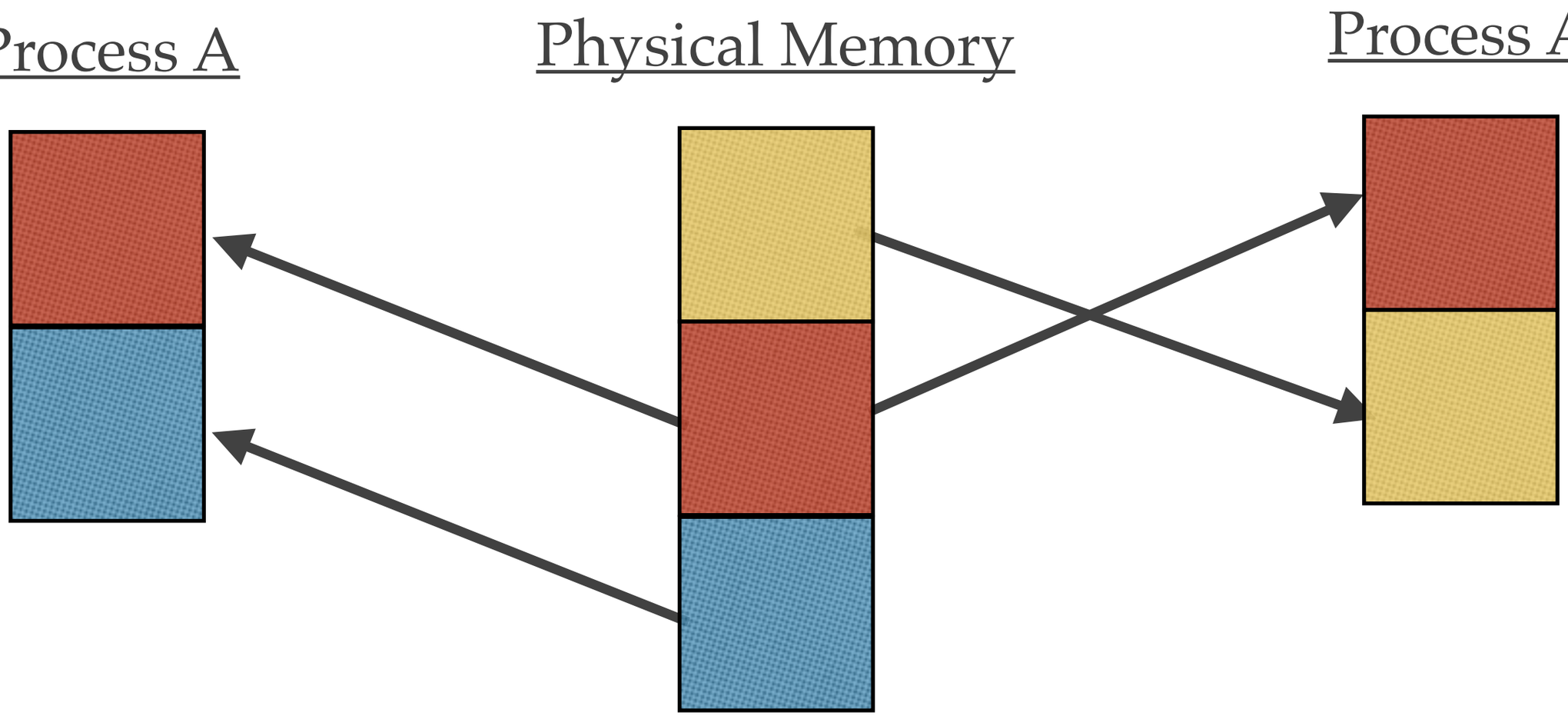
What are the two ways in which using multiple processes increases performance?

Mechanisms for Interprocess Communication

- Shared memory
  - POSIX
  - BSD mmap
- Message passing
  - Pipes, sockets, remote procedure calls

Shared Memory

- Processes see (i.e., share) the same physical memory



Shared Memory: POSIX Interface

- shm_get – returns the identifier of a shared memory segment
- shmat – attaches the shared memory segment to the address space
- shmdt – detaches the segment located at the specified address
- shmctl – control of shared memory segments, including deletion

Shared Memory: BSD Interface

- mmap with MAP_SHARED
  - Use backing file to share with unrelated processes
  - Use anonymous mapping to share with child processes
- munmap
  - Remove mmap mappings

mmap Interface

- void * mmap (void * addr, size_t length, int protection, int flags, int fd, off_t offset)
  - addr - Suggested address of where to place memory
  - length - The length of the shared memory segment
  - protection - The MMU access rights to use
  - flags - Controls lots of options
  - fd - File descriptor of backing file to use
  - offset - Offset within backing file to use

mmap Flags

Flag	Meaning
MAP_ANONYMOUS	Do not use a backing file
MAP_SHARED	Allow multiple processes to use the memory
MAP_FIXED	Refuse to allocate if addr is not available

What are some disadvantages of shared memory?

Message Passing

- Direct or indirect communication – processes or ports
- Fixed or variable size messages
- Send by copy or reference
- Automatic or explicit buffering
- Blocking or non-blocking (send or receive)

Examples of Message Passing

- Pipes
- Sockets
- Mach ports
- Windows 2000 Local Procedure Call
- Remote Procedure Call (e.g., RPC, CORBA)

Interprocess Communication: Pipes

- Conduit allowing two processes to communicate
  - Unidirectional or bidirectional
  - Full-duplex or half-duplex two-way communication
  - Is parent-child relationship required?
  - Is communication across a network allowed?

Unix Pipes

- A unidirectional data channel that can be used for interprocess communication
- Pipe ceases to exist once closed or when process terminates
- System calls
  - pipe (int fd[])
  - dup2

What are some disadvantages of message passing?

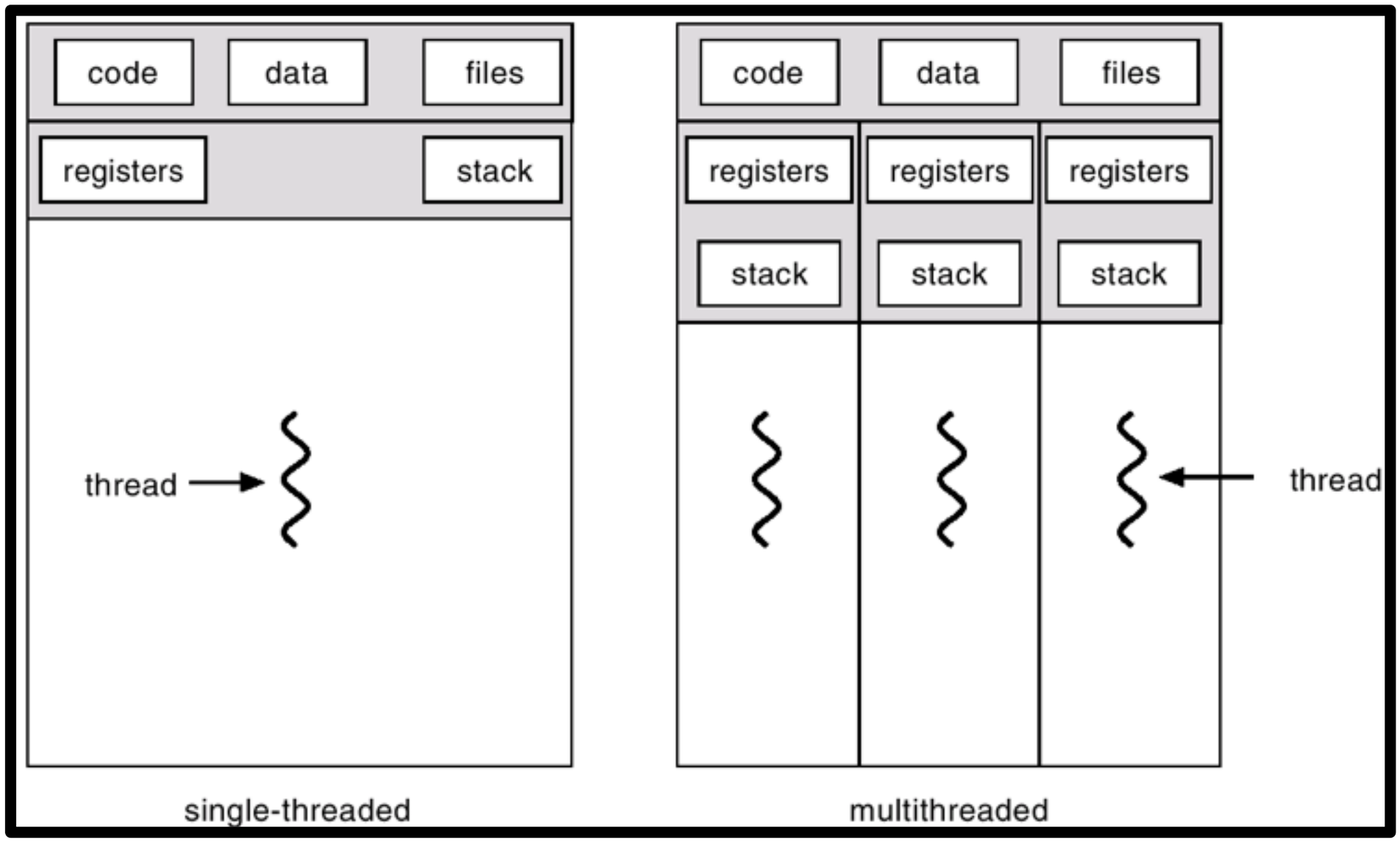
Threads

Processes or Threads

- A process or thread is a potentially-active execution context
  - It is a program *in execution*
- Threads can run
  - Truly in parallel (on multiple CPUs or cores)
  - Unpredictably interleaved (on a single CPU)
  - Run-until-block (coroutine-style)

Thread – a program in execution *without a dedicated address space*.

OS memory protection is only applied to processes.



Processes Vs. Threads

- Process
  - Single address space
  - Single thread of control for executing program
  - State: page tables, swap images, file descriptors, queued I/O requests
- Threads
  - Separate notion of execution from the rest of the definition of a process
  - Page tables, swap images, file descriptors, etc. potentially shared with other threads
  - Program counter, stack of activation records, control block (e.g., saved registers/state info for thread management)

Why Use Threads?

- Multithreading is used for parallelism/concurrency
- Why not multiple processes?
  - Memory sharing
  - Efficient synchronization between threads
  - Less context switch overhead

User/Kernel Threads

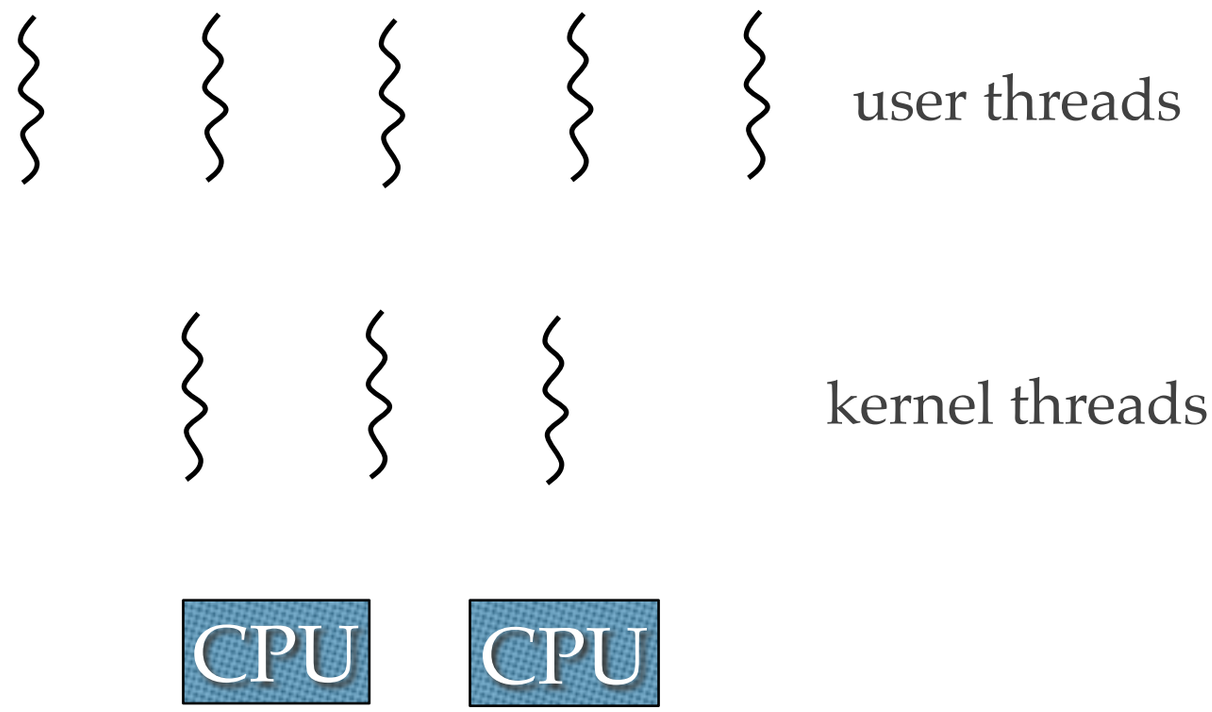
- User threads
  - Thread data structure is in user-mode memory
  - Scheduling/switching done at user mode
- Kernel threads
  - Thread data structure is in kernel memory
  - Scheduling/switching done by the OS kernel

User/Kernel Threads (cont.)

- Benefits of user threads
  - lightweight – less context switching overhead
  - more efficient synchronization??
  - flexibility – allow application-controlled scheduling
- Problems of user threads
  - can't use more than one processor
  - oblivious to kernel events, e.g., all threads in a process are put to wait when only one of them does I/O

Mixed User/Kernel Threads

- M user threads run on N kernel threads (M≥N)
  - N=1: pure user threads
  - M=N: pure kernel threads
  - M>N>1: mixed model



Solaris/Linux Threads

- Solaris
  - Supports mixed model
- Linux
  - No standard user threads on Linux
  - Processes and threads treated in a similar manner (both called tasks)
  - Processes are tasks with exclusive address space
  - Tasks can also share the address space, open files, …

Challenges with Threads

- Thread-local storage – what about globals?
- Stack management
  - Where do you put them?
  - How to detect when they run into each other?
- Heap
  - Usually shared
  - Memory allocator must be synchronized and reentrant

More Issues with Threads

- Interaction with fork() and exec() system calls
  - Two versions of fork()?
  - What happens with a thread calls exec()?
- Signal handling – which thread should the signal be delivered to?
  - Synchronous
  - All
  - Assigned thread
  - Unix: could assign a specific thread to handle signals
  - Windows: asynchronous procedure calls, which are thread-specific

Even More Challenges with Threads!

- Unix predates threads
- Many libraries and system calls assume single thread
  - Poster child: errno
- Many APIs now have reentrant versions: getpwnam_r()
- Restrictions on signal handlers

POSIX Threads API

POSIX Threads (Pthreads)

- Each OS has its own thread package with different Application Programming Interfaces ⇒ poor portability.
- Pthreads
  - A POSIX standard API for thread management and synchronization.
  - API specifies behavior of the thread library, not the implementation.
  - Commonly supported in Unix operating systems.

Pthreads: A Shared Memory Programming Model

- POSIX standard shared-memory multithreading interface
- Not just for parallel programming, but for general multithreaded programming
- Provides primitives for thread management and synchronization

What does the user have to do?

- Decide how to decompose the computation into parallel parts
- Create (and destroy) threads to support that decomposition
- Add synchronization to make sure dependences are covered

Thread Creation

```
int pthread_create (pthread_t *new_id,
                    const pthread_attr_t *attr,
                    void *(*func) (void *),
                    void *arg)
```

- new_id: thread's unique identifier
- attr: ignore for now
- func: function to be run in parallel
- arg: arguments for function func

Example of Thread Creation

```
void *func(void *arg) {
    int *I=arg;
    …
}

void main(){
    int X;
    pthread_t  id;
    …
    pthread_create(&id, NULL, func, &X);
    …
}
```

Pthread Termination

void pthread_exit(void *status)

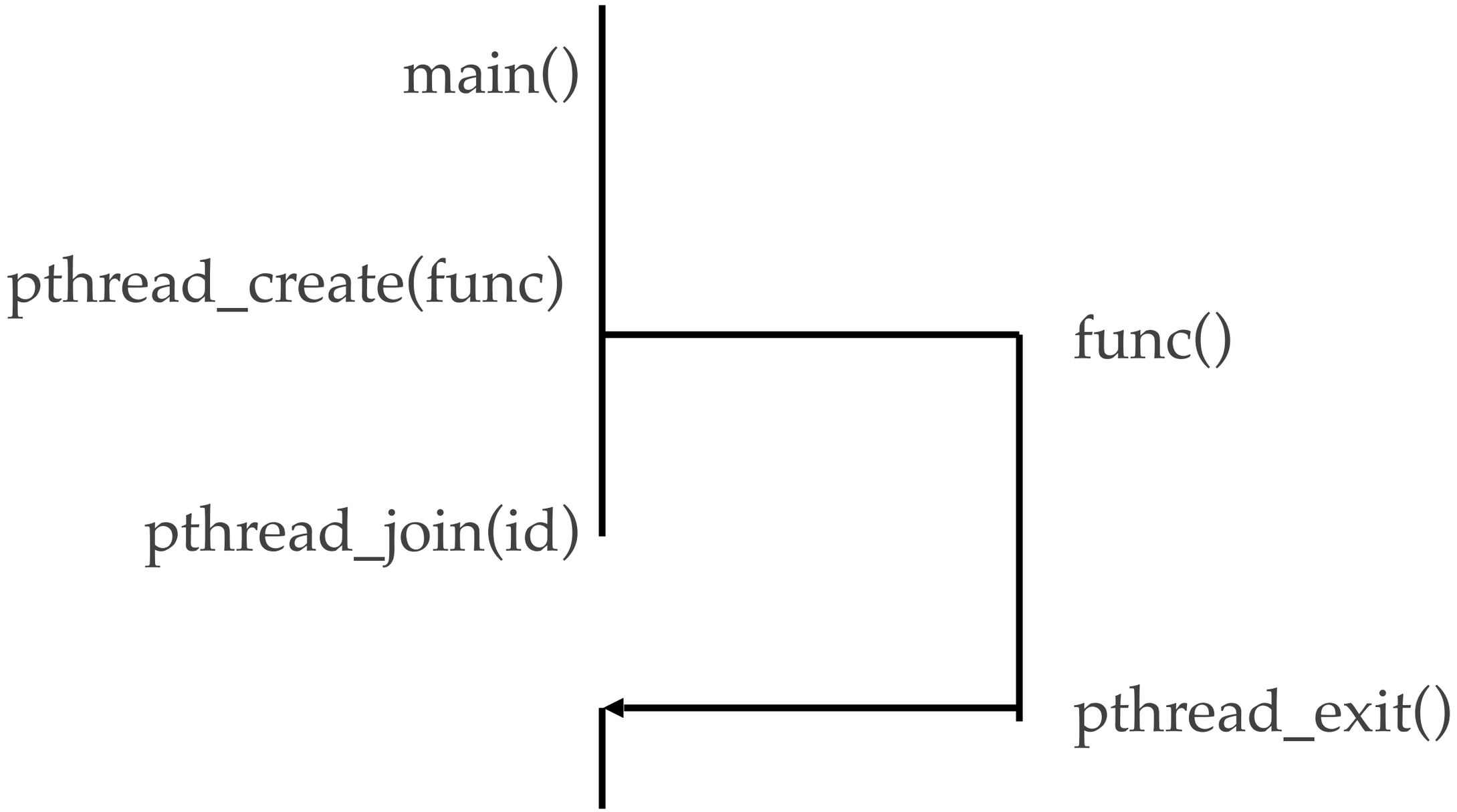
- Terminates the currently running thread.
- Is implicit when the function called in pthread_create() returns.

Thread Joining

```
int pthread_join (pthread_t new_id, void ** status)
```

- Waits for the thread with identifier new_id to terminate, either by returning or by calling pthread_exit().
- Status receives the return value or the value given as argument to pthread_exit().

Example of Thread Creation



pthread_join() looks awfully familiar. What is the equivalent for processes?

POSIX Thread Contention Scope

- Process contention scope
  - Thread library schedules user threads onto light-weight processes (kernel-level threads)
    - Use priority as defined by user – no preemption of threads with same priority
- System contention scope
  - Assign thread to its own kernel thread
  - Compete with all tasks

POSIX Thread Contention Scope

- Functions for setting/getting contention scope
  - pthread_attr_setscope()
  - pthread_attr_getscope()
- Values for contention scope
  - PTHREAD_SCOPE_PROCESS
  - PTHREAD_SCOPE_SYSTEM

Pthread Attributes

- pthread_attr_init(pthread_attr_t *attr), destroy – initializes attr to default value
  - Scope – pthread_attr_setscope (&attr, SCOPE)
  - Stack size – pthread_attr_getstacksize, pthread_attr_setstacksize
  - Priority
  - Joinable or detached

General Thread Structure

- Typically, a thread is a concurrent execution of a function or a procedure
- So, your program needs to be restructured such that parallel parts form separate procedures or functions

General Program Structure

- Encapsulate parallel parts in functions.
- Use function arguments to parametrize what a particular thread does.
- Call pthread_create() with the function and arguments, save thread identifier returned.
- Call pthread_join() with that thread identifier.

Pthreads Synchronization

- Create/exit/join
  - provide some form of synchronization
  - at a very coarse level
  - requires thread creation/destruction
- Need for finer-grain synchronization
  - mutex locks, reader-writer locks, condition variables, semaphores

Credits

- This slide is based on contributions from Sandhya Dwarkadas