Processes and Threads

Processes and threads

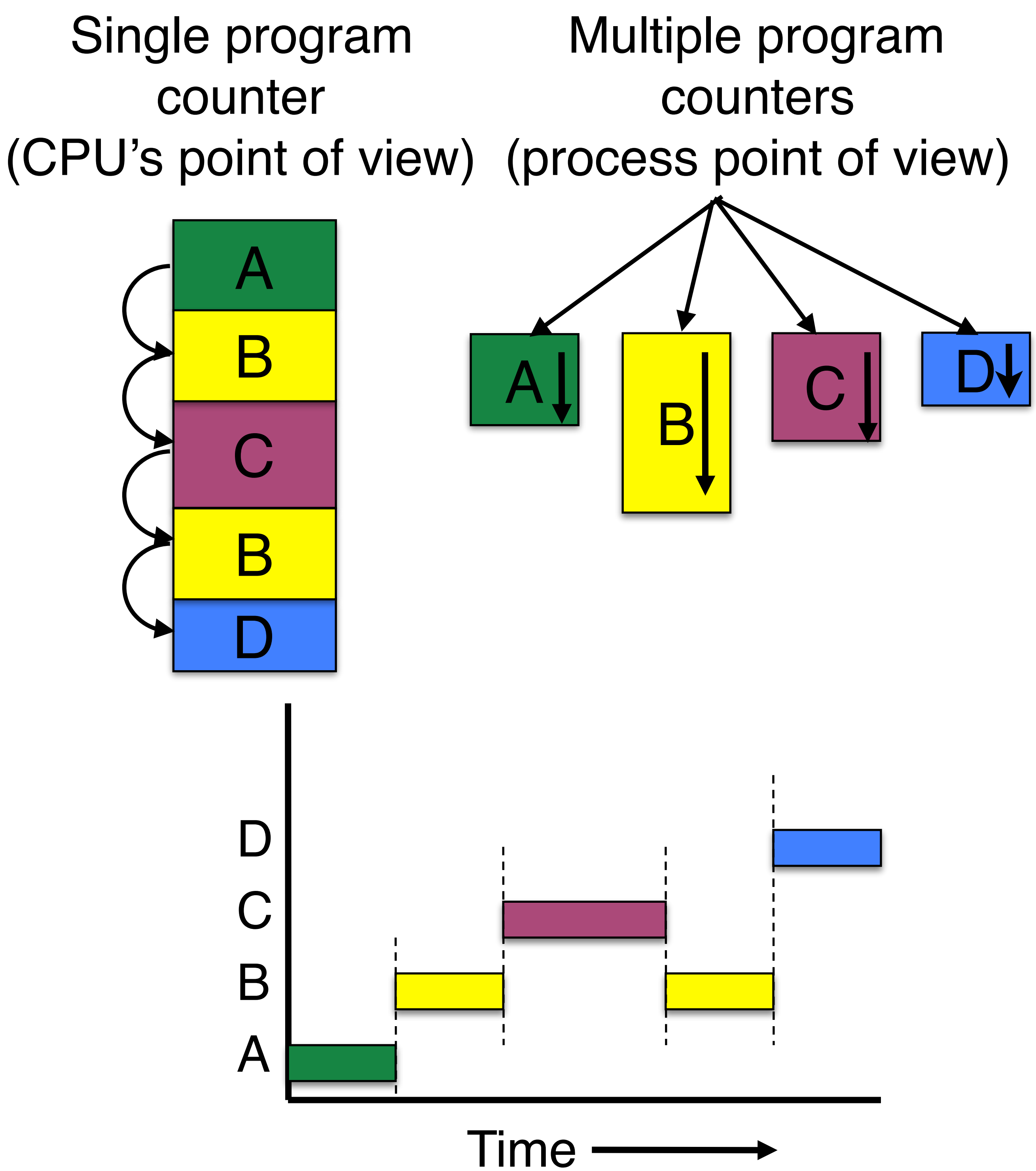
- Processes
- Threads
- Scheduling
- Interprocess communication (IPC)
- Classical IPC problems

What is a process?

- Code, data, and stack
  - Usually (but not always) has its own address space
- Program state
  - CPU registers
  - Program counter (current location in the code)
  - Stack pointer
- Only one process can be running in a single CPU core at any given time!
  - Multi-core CPUs can support multiple processes

The process model

- Multiprogramming of four programs
- Conceptual model
  - 4 independent processes
  - Processes run sequentially
- Only one program active at any instant!
  - That instant can be very short…
  - Only applies if there's a single CPU (with a single core) in the system



When is a process created?

- Processes can be created in two ways
  - System initialization: one or more processes created when the OS starts up
  - Execution of a process creation system call: something explicitly asks for a new process
- System calls can come from
  - User request to create a new process (system call executed from user shell)
  - Already running processes
    - User programs
    - System daemons

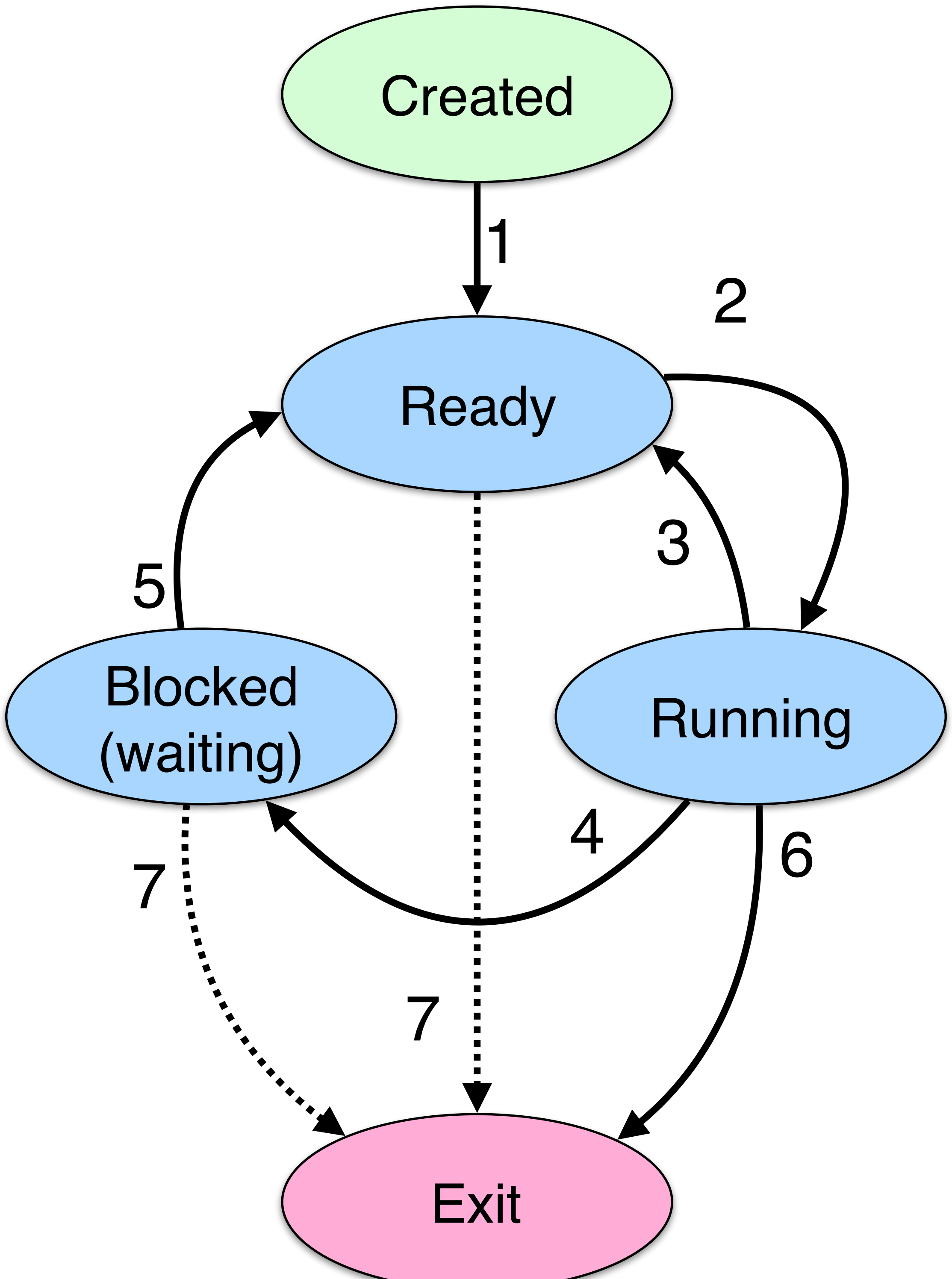
When do processes end?

- Conditions that terminate processes can be
  - Voluntary
  - Involuntary
- Voluntary
  - Normal exit
  - Error exit
- Involuntary
  - Fatal error (only sort of involuntary)
  - Killed by another process

Process hierarchies

- Parent creates a child process
  - Child processes can create their own children
- Forms a hierarchy
  - UNIX calls this a “process group”
  - If a process terminates, its children are “inherited” by the terminating process’s parent
- Windows has process groups
  - Multiple processes grouped together
  - One process is the “group leader”

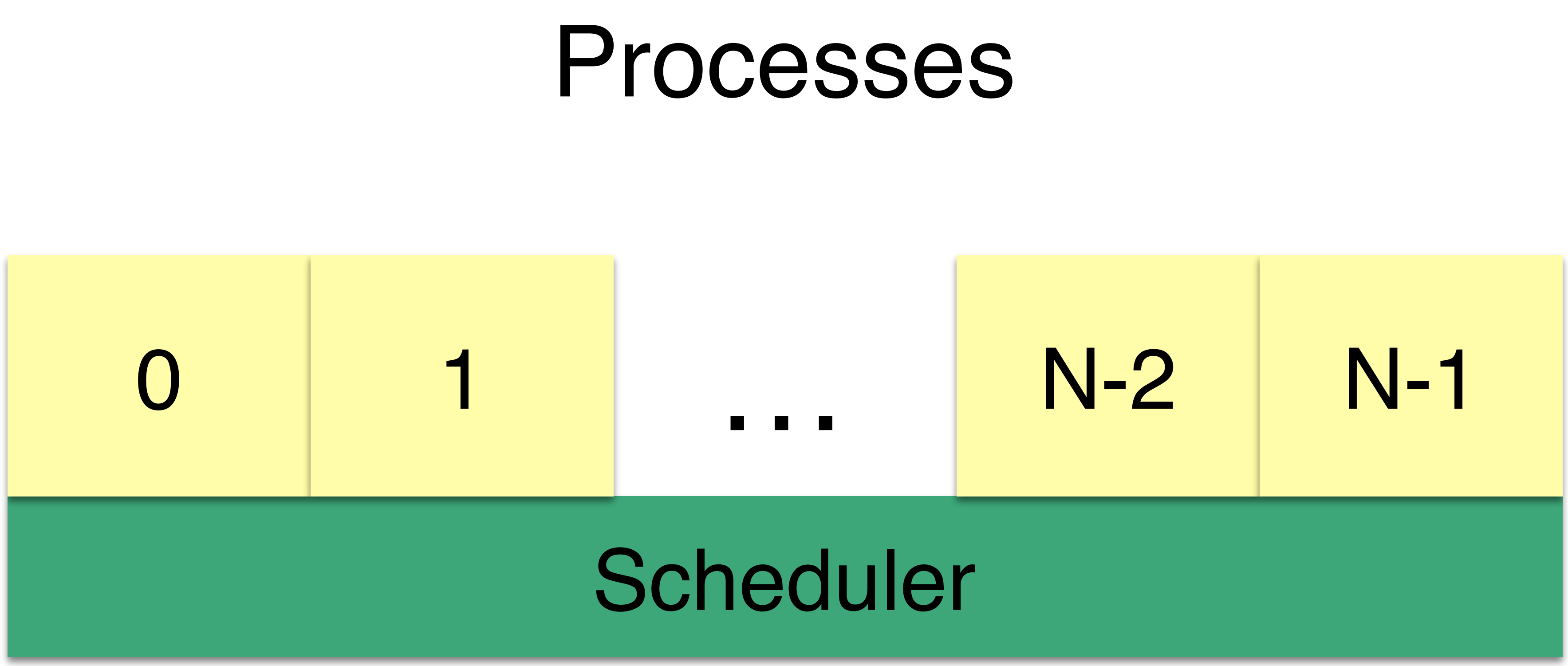
Process states



- Process in one of 5 states
  - Created
  - Ready
  - Running
  - Blocked
  - Exit
- Transitions between states
  1. Process enters ready queue
  2. Scheduler picks this process
  3. Scheduler picks a different process
  4. Process waits for an event such as I/O
  5. Event occurs
  6. Process exits
  7. (Process ended by another process)

Processes in the OS

- Two “layers” for processes
- Lowest layer of process-structured OS handles interrupts, scheduling
- Above that layer are sequential processes
  - Processes tracked in the process table
  - Each process has a process table entry



What's in a process table entry?

May be stored on stack: $\left\{ \begin{array}{l} \text{Registers} \\ \text{Program counter} \\ \text{CPU status word} \end{array} \right.$

Process management

Registers
Program counter
CPU status word
Stack pointer
Process state
Priority / scheduling parameters
Process ID
Parent process ID
Signals
Process start time
Total CPU usage

File management

Root directory
Working (current) directory
File descriptors
User ID
Group ID

Memory management

Pointers to text, data, stack

- or -

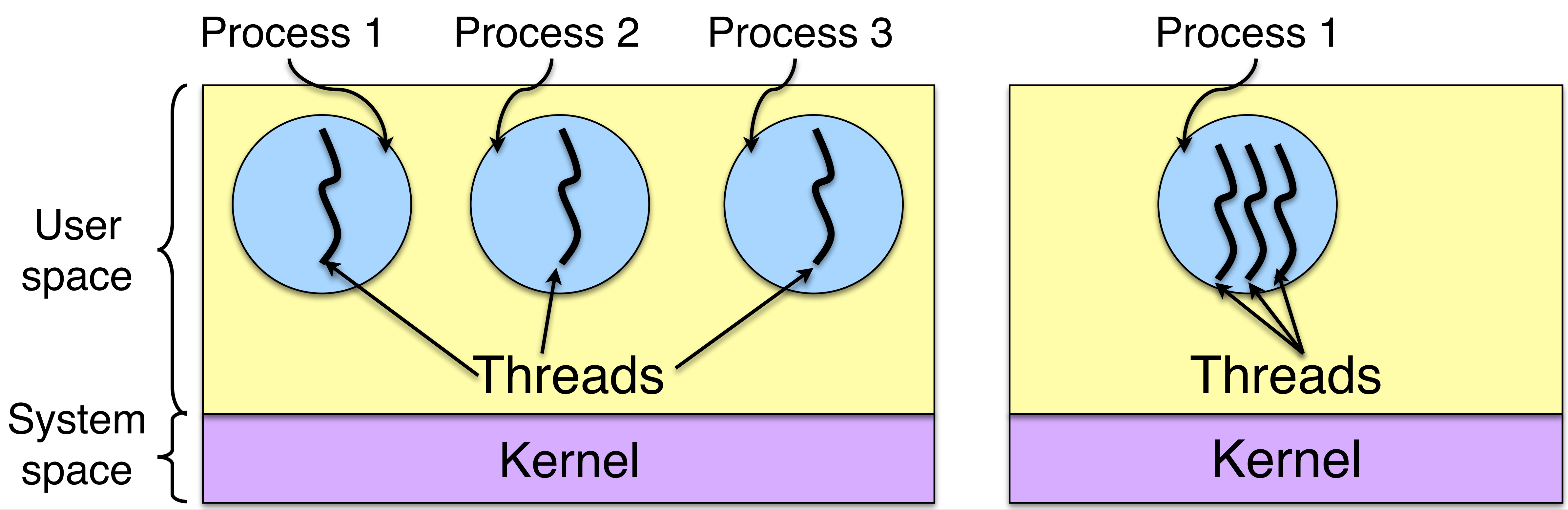
Pointer to page table

What happens on a trap/interrupt?

1. Hardware saves program counter (on stack or in a special register)
2. Hardware loads new PC, identifies interrupt
3. Assembly language routine saves registers
4. Assembly language routine sets up stack
5. Assembly language calls C to run service routine
6. Service routine calls scheduler
7. Scheduler selects a process to run next (might be the one interrupted…)
8. Assembly language routine loads PC & registers for the selected process

Threads: “processes” sharing memory

- Process ⇔ address space
- Thread ⇔ program counter / stream of instructions
- Two examples
  - Three processes, each with one thread
  - One process with three threads



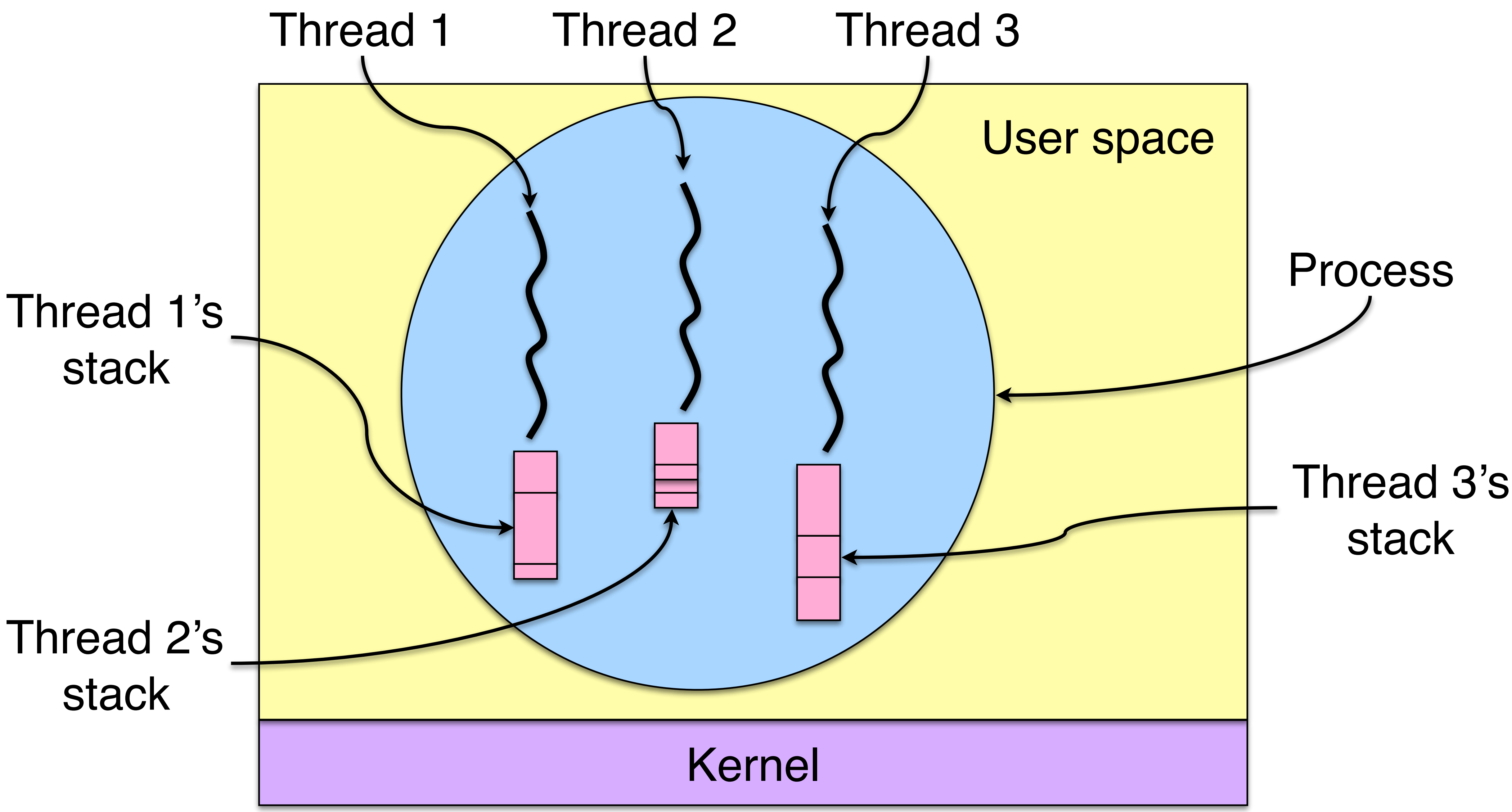
Process & thread information

Per process items

Address space
Open files
Child processes
Signals & handlers
Accounting info
Global variables

Per thread items	Per thread items	Per thread items
Program counter Registers Stack & stack pointer State (local variables)	Program counter Registers Stack & stack pointer State (local variables)	Program counter Registers Stack & stack pointer State (local variables)

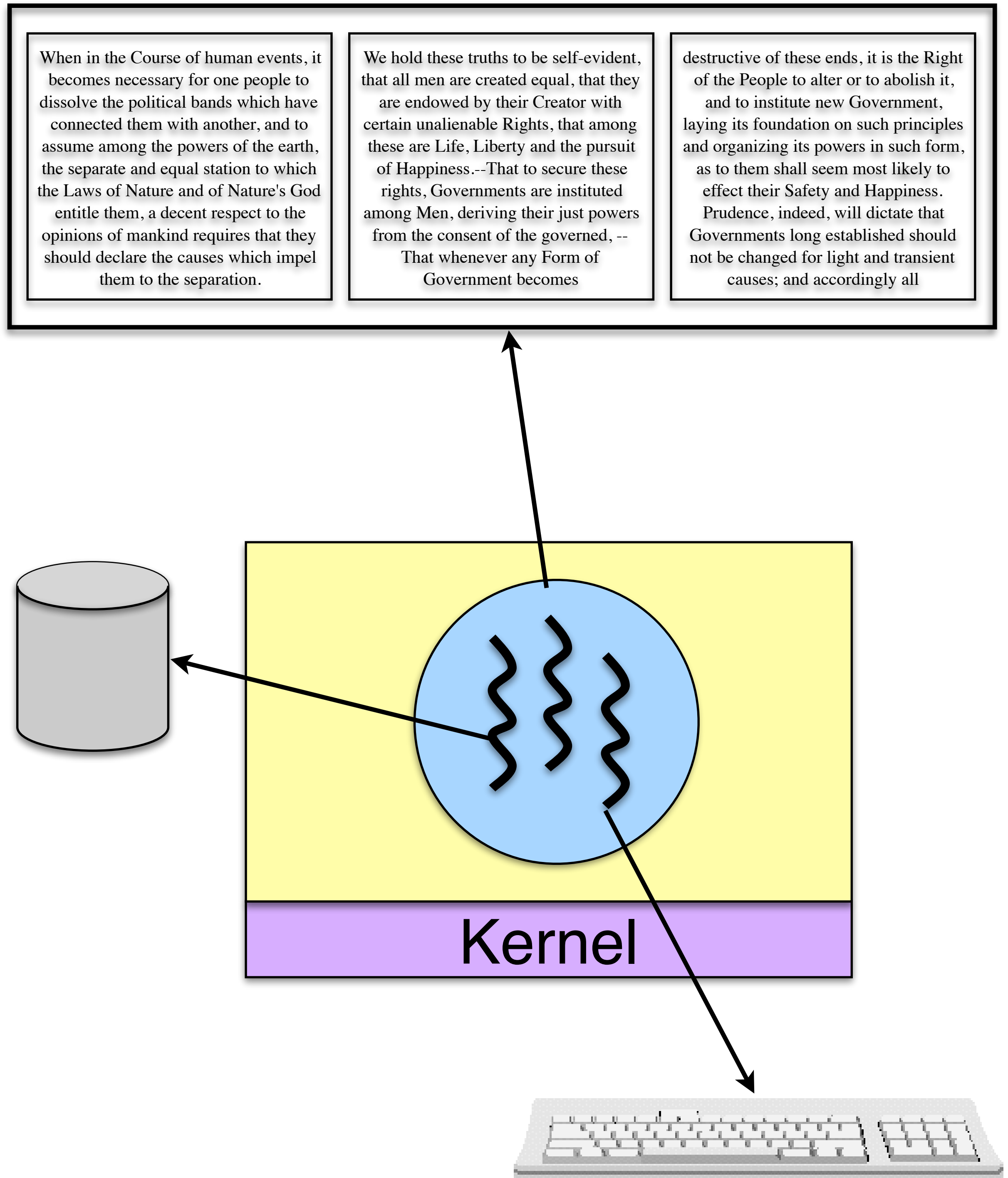
Threads & stacks



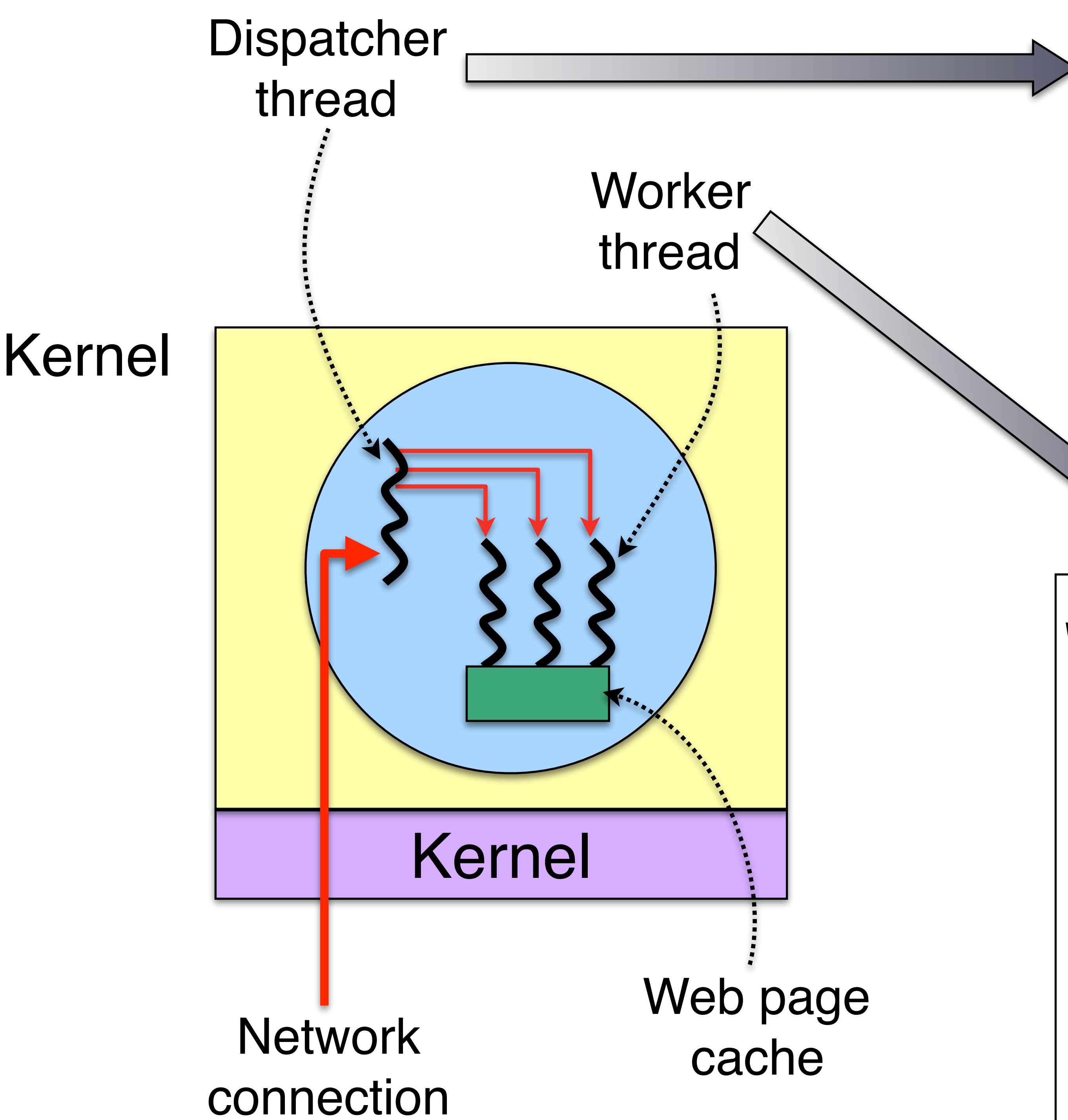
➡ Each thread has its own stack!

Why use threads?

- Allow a single application to do many things at once
  - Simpler programming model
  - Less waiting
- Threads are faster to create or destroy
  - No separate address space
- Overlap computation and I/O
  - Could be done without threads, but it's harder
- Example: word processor
  - Thread to read from keyboard
  - Thread to format document
  - Thread to write to disk



Multithreaded Web server




```
while(TRUE) {
  getNextRequest(&buf);
  handoffWork(&buf);
}
```

```
while(TRUE) {
  waitForWork(&buf);
  lookForPageInCache(&buf,&page);
  if(pageNotInCache(&page)) {
    readPageFromDisk(&buf,&page);
  }
  returnPage(&page);
}
```

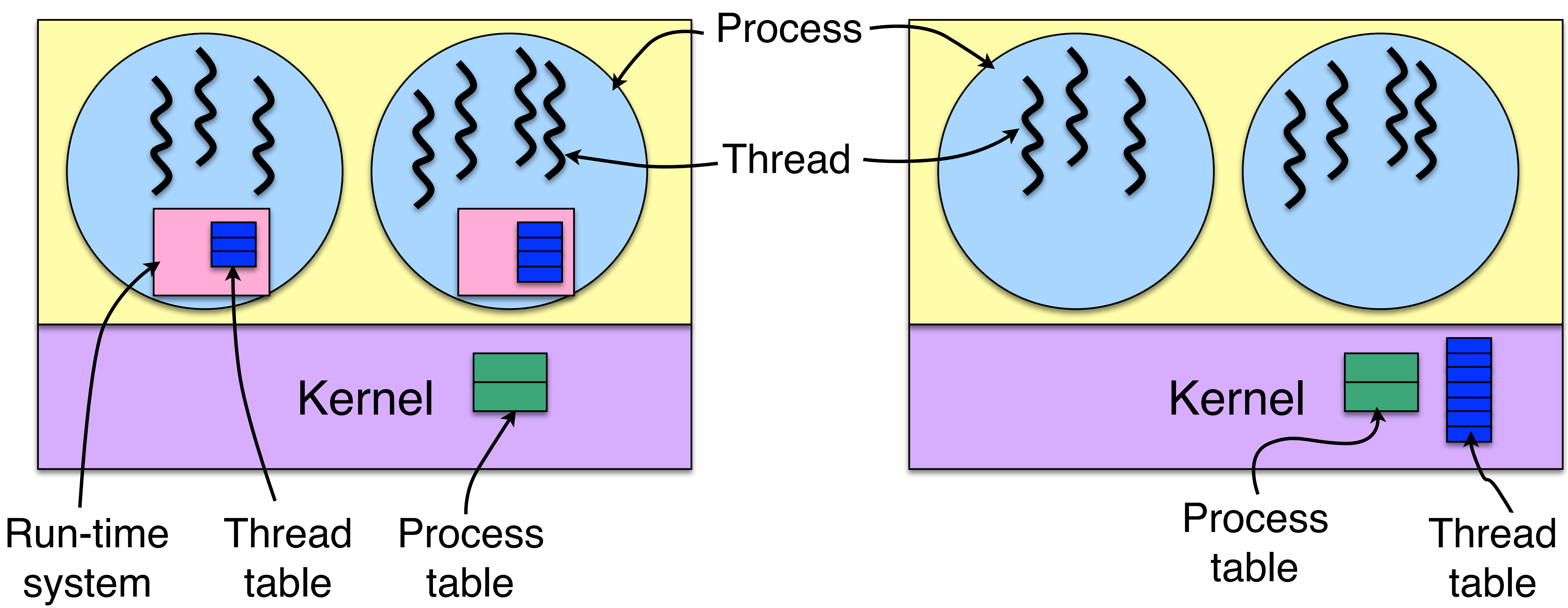
Three ways to build a server

- Multithreaded server
  - Parallelism
  - Blocking system calls
  - May use pop-up threads: create a new thread in response to an incoming message (rather than reusing a thread)
- Single-threaded process: slow, but easier to do
  - No parallelism
  - Blocking system calls
- Finite-state machine (event model)
  - Each activity has its own state: states change when system calls complete or interrupts occur
  - Parallelism
  - Nonblocking system calls

Issues with using threads

- May be tricky to convert single-threaded code to multithreaded code
- Re-entrant code
  - Code must function properly when multiple threads are using it simultaneously
  - Need to be careful when using static or global variables
    - Returned structures
    - Buffers
- Error management
  - What happens when just a single thread has an error?
  - Can’t simply kill the process, since other threads might be running

Implementing threads



User-level threads

+ No need for kernel support
– May be slower than kernel threads
– Harder to do non-blocking I/O

Kernel-level threads

+ More flexible scheduling
+ Non-blocking I/O
– Not (necessarily) portable

POSIX threads

- Standard interface to threading library
- May be implemented in either user or kernel space
  - Some operating systems provide support for both!
- Allows thread-based programs to be portable

Thread call `(Pthread_xx)`	**Description**
create	Create a new thread
exit	Terminate the calling thread
join	Wait for a specific thread to exit
yield	Release the CPU, allowing another thread to run

Processes & threads in Linux

- Supports POSIX standard
- Linux supports kernel-level threads (lightweight processes)
  - Share address space, file descriptors, etc.
  - Each has its own process descriptor in memory
- Linux processes (incl. lightweight) all have unique identifiers
  - Threads sharing address space are grouped into process groups
  - Identifier shared by the group is that of the leader
- Each process has its own 8KB region that stores
  - Kernel stack
    - Kernel has a small stack: about 4KB!
  - Low-level thread information
- Other information stored in a separate data structure
  - Memory allocated to the process
  - Open files
  - Signal information

Processes & threads in FreeBSD

- Supports POSIX threads (as does Linux)
- Processes are heavyweight
  - Unique process ID
  - Individual address space
  - List of associated threads
- Threads are “variable-weight”
  - Must contain thread control block, thread kernel stack, and thread state
  - Lightest weight
    - Share *all* other process resources!
    - Thread library keeps track of user-level stacks
    - Might be useful for “thread pool” implementations
  - Heavier weight: created by rfork()
    - Share fewer resources
    - May have separate stack (likely) and data (possibly) spaces
    - Gets its own process ID
    - Still shares some address space, including global variables