What is a Process?

Answer 1: a process is an abstraction of a program in execution

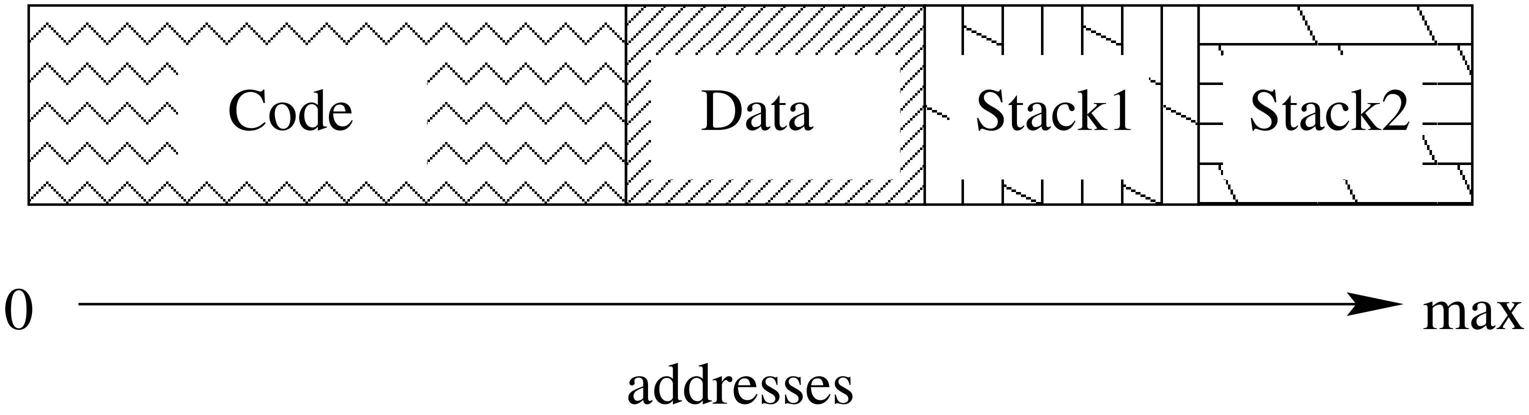
Answer 2: a process consists of

- an address space
- a thread of execution (possibly several threads)
- other resources associated with the running program. For example:
  - open files
  - sockets
  - attributes, such as a name (process identifier)
  - ...

A process with one thread is a *sequential* process. A process with more than one thread is a *concurrent* process.

What is an Address Space?

- For now, think of an address space as a portion of the primary memory of the machine that is used to hold the code, data, and stack(s) of the running program.
- For example:



- We will elaborate on this later.

What is a Thread?

- A thread represents the control state of an executing program.
- Each thread has an associated *context*, which consists of
  - the values of the processor's registers, including the program counter (PC) and stack pointer
  - other processor state, including execution privilege or mode (user/system)
  - a stack, which is located in the address space of the thread's process

The Operating System and the Kernel

- We will use the following terminology:

  kernel: The operating system kernel is the part of the operating system that responds to system calls, interrupts and exceptions.

  operating system: The operating system as a whole includes the kernel, and may include other related programs that provide services for applications. This may include things like:

  - utility programs
  - command interpreters
  - programming libraries

The OS Kernel

- Usually kernel code runs in a privileged execution mode, while the rest of the operating system does not.
- The kernel is a program. It has code and data like any other program.
- For now, think of the kernel as a program that resides in its own address space, separate from the address spaces of processes that are running on the system. Later, we will elaborate on the relationship between the kernel's address space and process address spaces.

Kernel Privilege, Kernel Protection

- What does it mean to run in privileged mode?
- Kernel uses privilege to
  - control hardware
  - protect and isolate itself from processes
- privileges vary from platform to platform, but may include:
  - ability to execute special instructions (like `halt`)
  - ability to manipulate processor state (like execution mode)
  - ability to access virtual addresses that can't be accessed otherwise
- kernel ensures that it is *isolated* from processes. No process can execute or change kernel code, or read or write kernel data, except through controlled mechanisms like system calls.

System Calls

- System calls are the interface between processes and the kernel.
- A process uses system calls to request operating system services.
- From point of view of the process, these services are used to manipulate the abstractions that are part of its execution environment. For example, a process might use a system call to
  - open a file
  - send a message over a pipe
  - create another process
  - increase the size of its address space

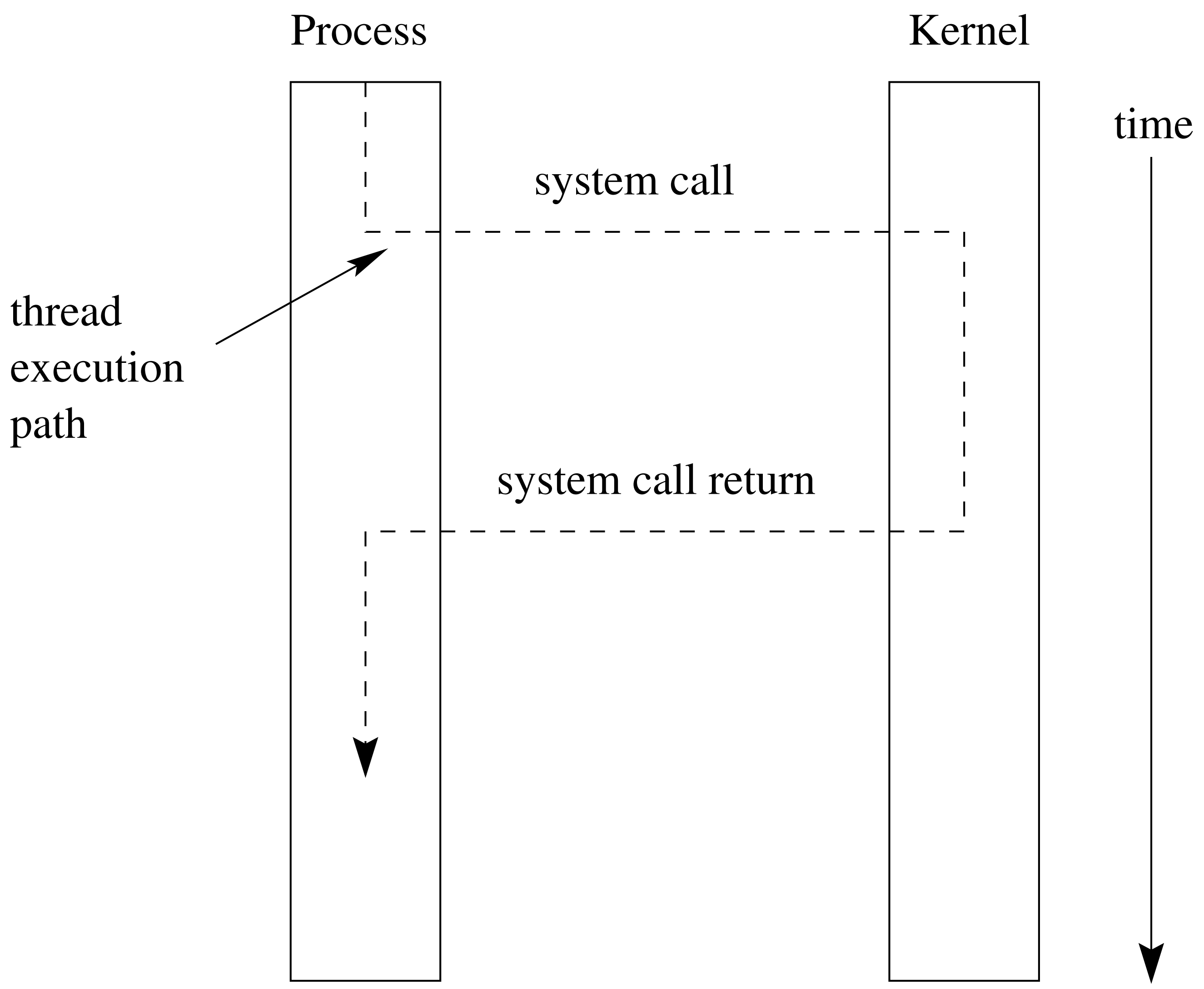
How System Calls Work

- The hardware provides a mechanism that a running program can use to cause a system call. Often, it is a special instruction, e.g., the MIPS `syscall` instruction.
- What happens on a system call:
  - key parts of the current thread context, like the program counter and the stack pointer, are saved
  - the processor is switched to system (privileged) execution mode
  - the thread context is changed so that:
    - the program counter is set to a fixed (determined by the hardware) memory address, which is within the kernel's address space
    - the stack pointer is pointed at a stack in the kernel's address space

System Call Execution and Return

- Once a system call occurs, the calling thread will be executing a system call handler, which is part of the kernel, in system mode.
- The kernel's handler determines which service the calling process wanted, and performs that service.
- When the kernel is finished, it returns from the system call. This means:
  - switch the processor back to unprivileged (user) execution mode
  - restore the key parts of the thread context that were saved when the system call was made
- Now the thread is executing the calling process' program again, picking up where it left off when it made the system call.

System Call Diagram



Exceptions

- Exceptions are another way that control is transferred from a process to the kernel.
- Exceptions are conditions that occur during the execution of an instruction by a process. For example:
  - arithmetic error, e.g, overflow
  - illegal instruction
  - memory protection violation
  - page fault (to be discussed later)
- exceptions are detected by the hardware

Exceptions (cont'd)

- when an exception occurs, control is transferred (by the hardware) to a fixed address in the kernel
- transfer of control happens in much the same way as it does for a system call. (In fact, a system call can be thought of as a type of exception, and they are sometimes implemented that way.)
- in the kernel, an exception handler determines which exception has occurred and what to do about it. For example, it may choose to destroy a process that attempts to execute an illegal instruction.

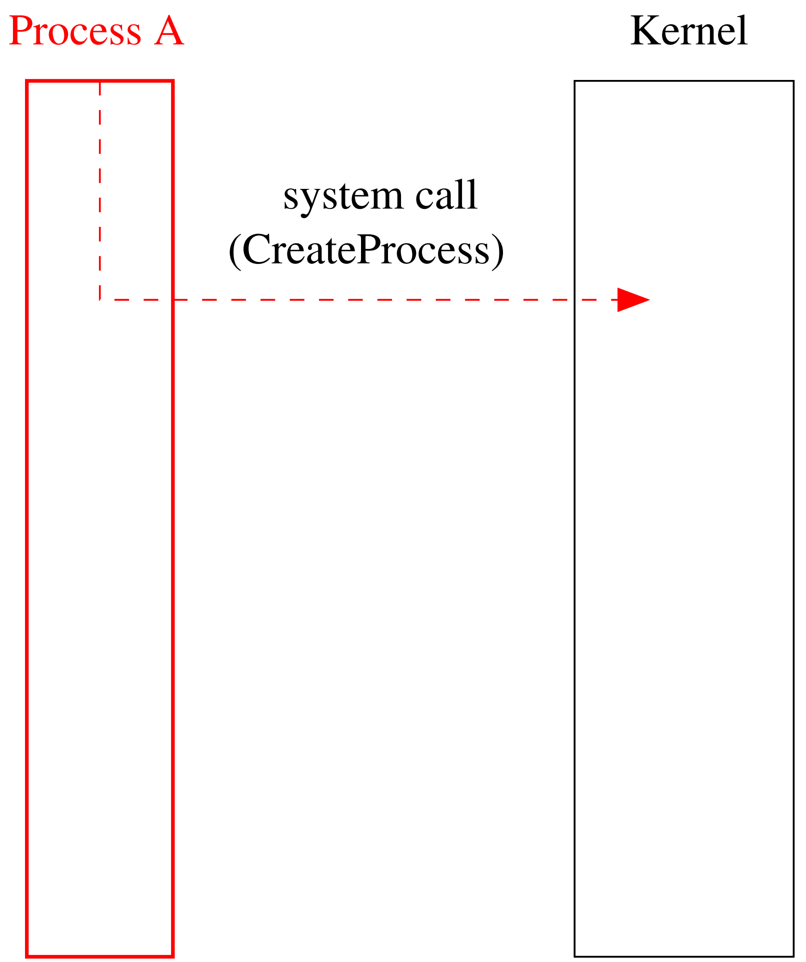
Interrupts

- Interrupts are a third mechanism by which control may be transferred to the kernel
- Interrupts are similar to exceptions. However, they are caused by hardware devices, not by the execution of a program. For example:
  - a network interface may generate an interrupt when a network packet arrives
  - a disk controller may generate an interrupt to indicate that it has finished writing data to the disk
  - a timer may generate an interrupt to indicate that time has passed
- Interrupt handling is similar to exception handling - current execution context is saved, and control is transferred to a kernel interrupt handler at a fixed address.

Multiprogramming

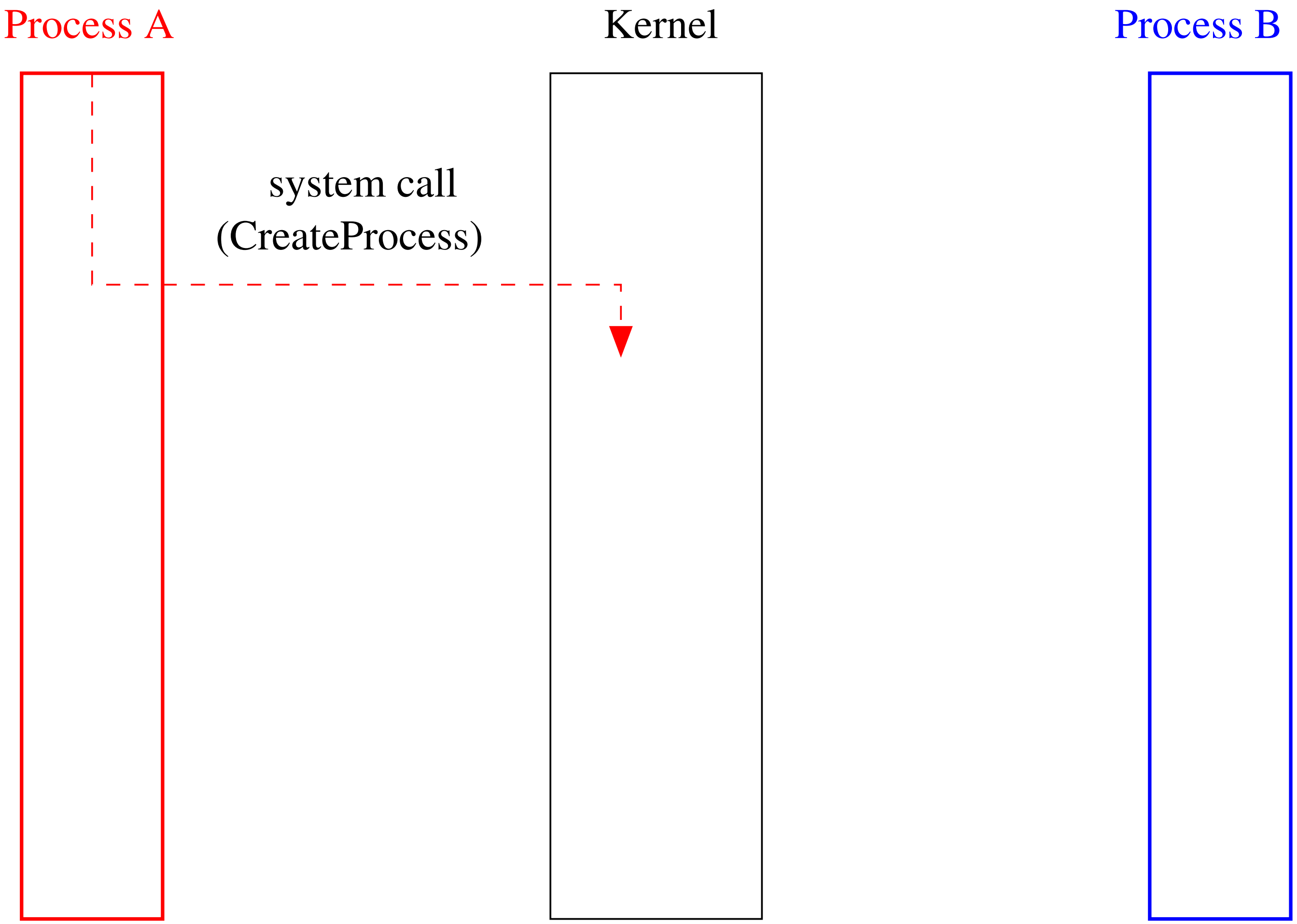
- multiprogramming means having multiple processes existing at the same time
- most modern, general purpose operating systems support multiprogramming
- all processes share the available hardware resources, with the sharing coordinated by the operating system:
  - Each process uses some of the available memory to hold its address space. The OS decides which memory and how much memory each process gets
  - OS can coordinate shared access to devices (keyboards, disks), since processes use these devices indirectly, by making system calls.
  - Processes *timeshare* the processor(s). Again, timesharing is controlled by the operating system.
- OS ensures that processes are isolated from one another. Interprocess communication should be possible, but only at the explicit request of the processes involved.

Process Creation Multiprogramming Example (Step 1)



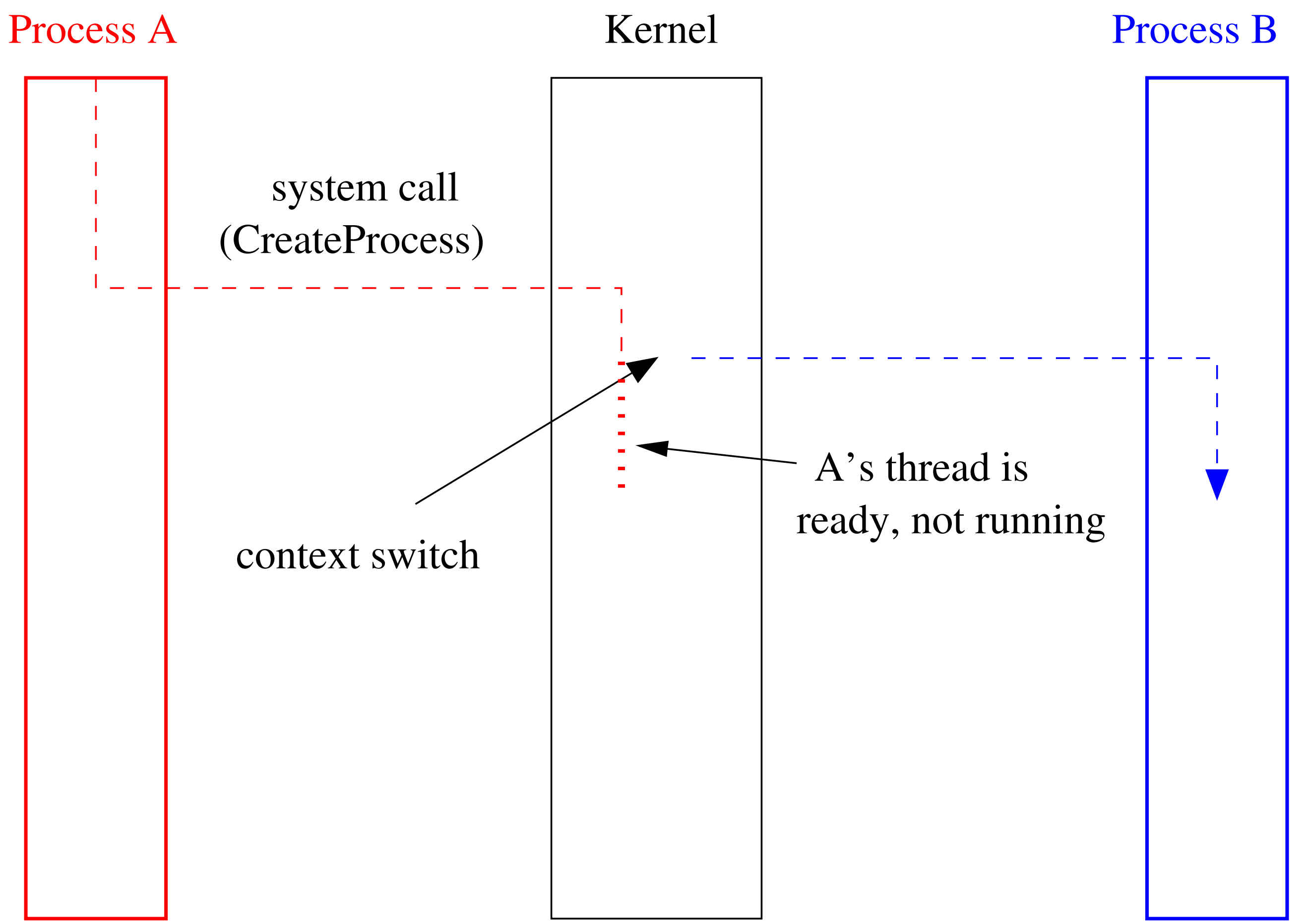
Parent process (Process A) requests creation of a new process.

Process Creation Multiprogramming Example (Step 2)



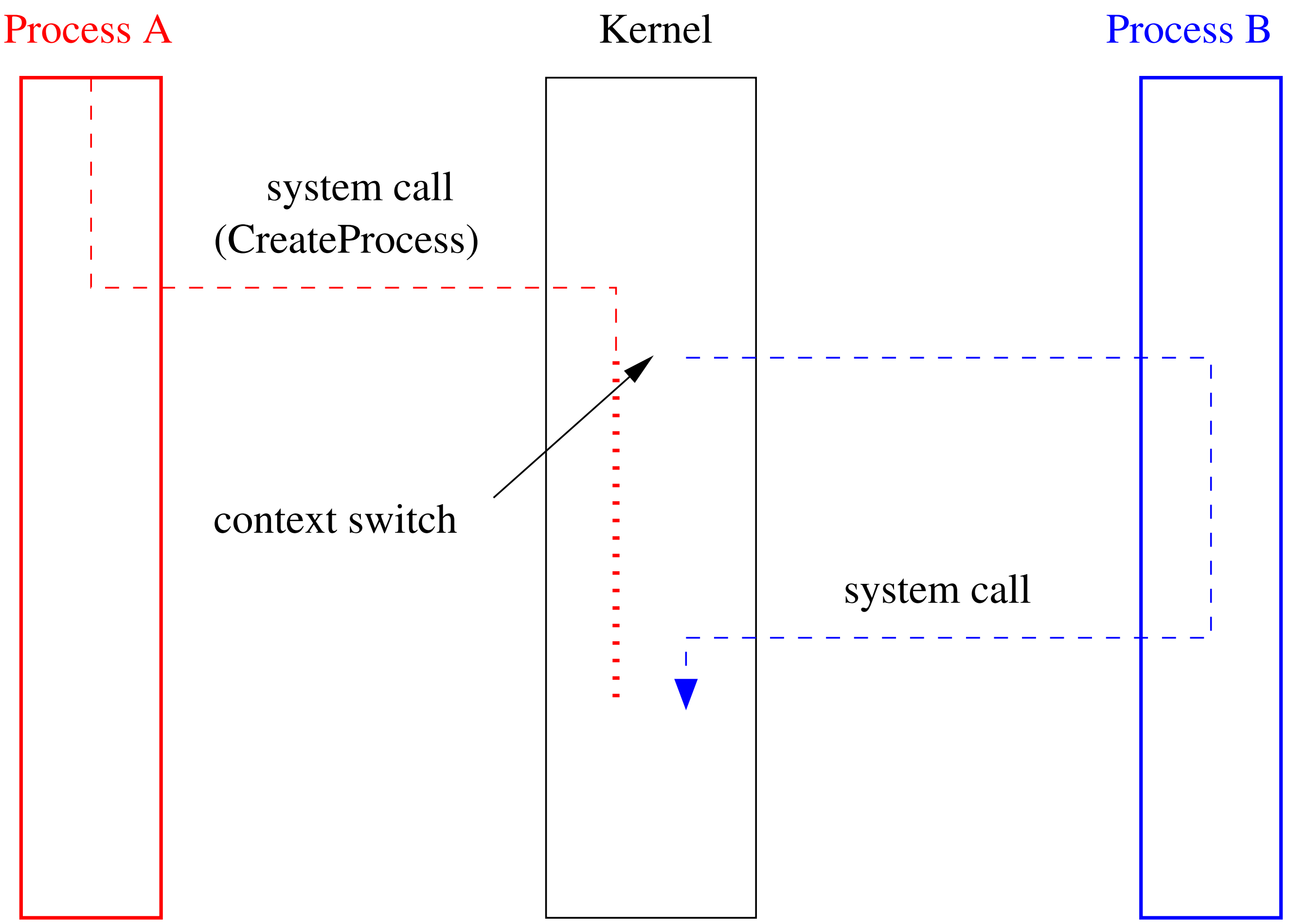
Kernel creates child process (Process B).

Process Creation Multiprogramming Example (Step 3)



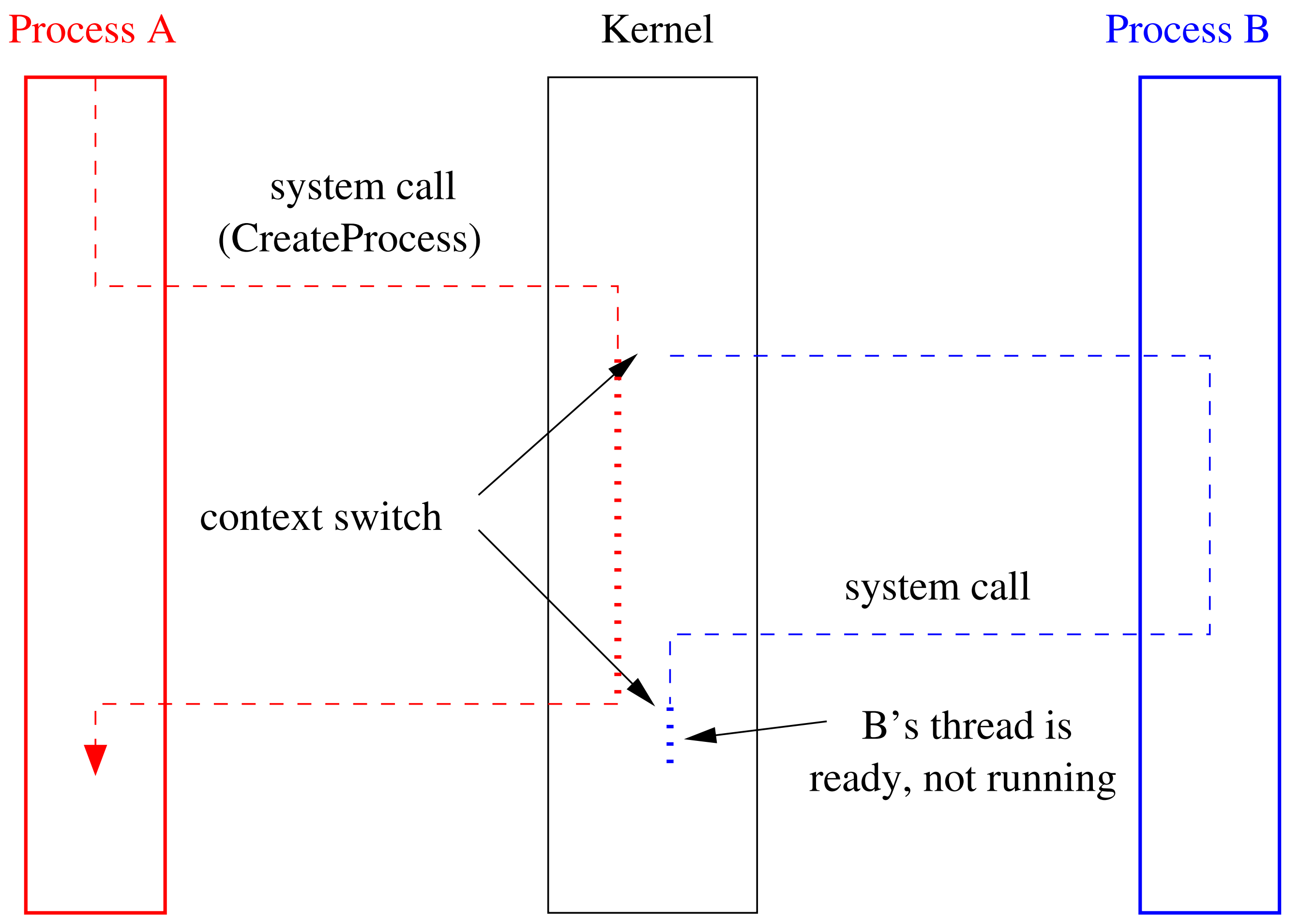
Kernel allows child process to run, switches execution context to Process B.

Process Creation Multiprogramming Example (Step 4)



Process B later makes a system call.

Process Creation Multiprogramming Example (Step 5)



Kernel decides Process A will run, switches context again.

Process Interface

- A running programs may use process-related system calls to manipulate its own process, or other processes in the system.
- The process interface will usually include:

Creation: make new processes, e.g., `Exec` in Nachos

Destruction: terminate a process, e.g., `Exit` in Nachos

Synchronization: wait for some event, e.g., `Join` in Nachos

Attribute Mgmt: read or change process attributes, such as the process identifier or owner or scheduling priority

The Process Model

- Although the general operations supported by the process interface are straightforward, there are some less obvious aspects of process behaviour that must be defined by an operating system.

  Process Initialization: When a new process is created, how is it initialized? What is in the address space? What is the initial thread context? Does it have any other resources?

  Multithreading: Are concurrent processes supported, or is each process limited to a single thread.

  Inter-Process Relationships: Are there relationships among processes, e.g, parent/child? If so, what do these relationships mean?

Implementation of Processes

- The kernel maintains information about all of the processes in the system in a data structure often called the process table.
- Information about individual processes is stored in a structure that is sometimes called a *process control block (PCB)*. In practice, however, information about a process may not all be located in a single data structure.
- Per-process information may include:
  - process identifier and owner
  - current process state and other scheduling information
  - lists of available resources, such as open files
  - accounting information
  - and more . . ....

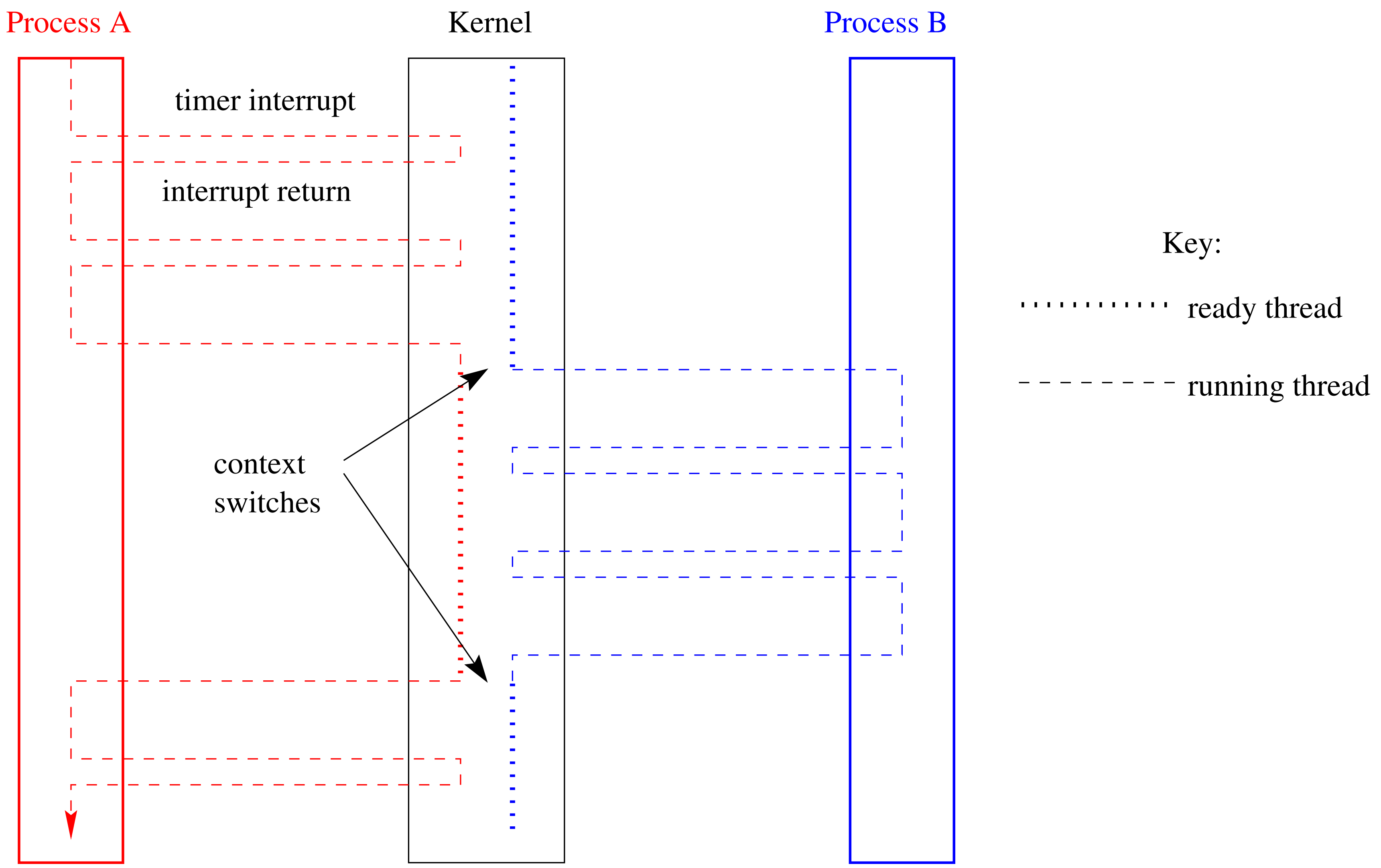
Processor Scheduling Basics

- Only one thread at a time can run on a processor.
- Processor scheduling means deciding how threads should share the available processor(s)
- Round-robin is a simple *preemptive* scheduling policy:
  - the kernel maintains a list of *ready* threads
  - the first thread on the list is *dispatched* (allowed to run)
  - when the running thread has run for a certain amount of time, called the scheduling quantum, it is *preempted*
  - the preempted thread goes to the back of the ready list, and the thread at the front of the list is dispatched.
- More on scheduling policies later.

Implementing Preemptive Scheduling

- The kernel uses interrupts from the system timer to measure the passage of time and to determine whether the running process's quantum has expired.
- All interrupts transfer control from the running program to the kernel.
- In the case of a timer interrupt, this transfer of control gives the kernel the opportunity to preempt the running thread and dispatch a new one.

Using Timer Interrupts to Timeshare (Example)



Blocked Threads

- Sometimes a thread will need to wait for an event. Examples:
  - wait for data from a (relatively) slow disk
  - wait for input from a keyboard
  - wait for another thread to leave a critical section
  - wait for busy device to become idle
- The OS scheduler should only allocate the processor to threads that are not blocked, since blocked threads have nothing to do while they are blocked.

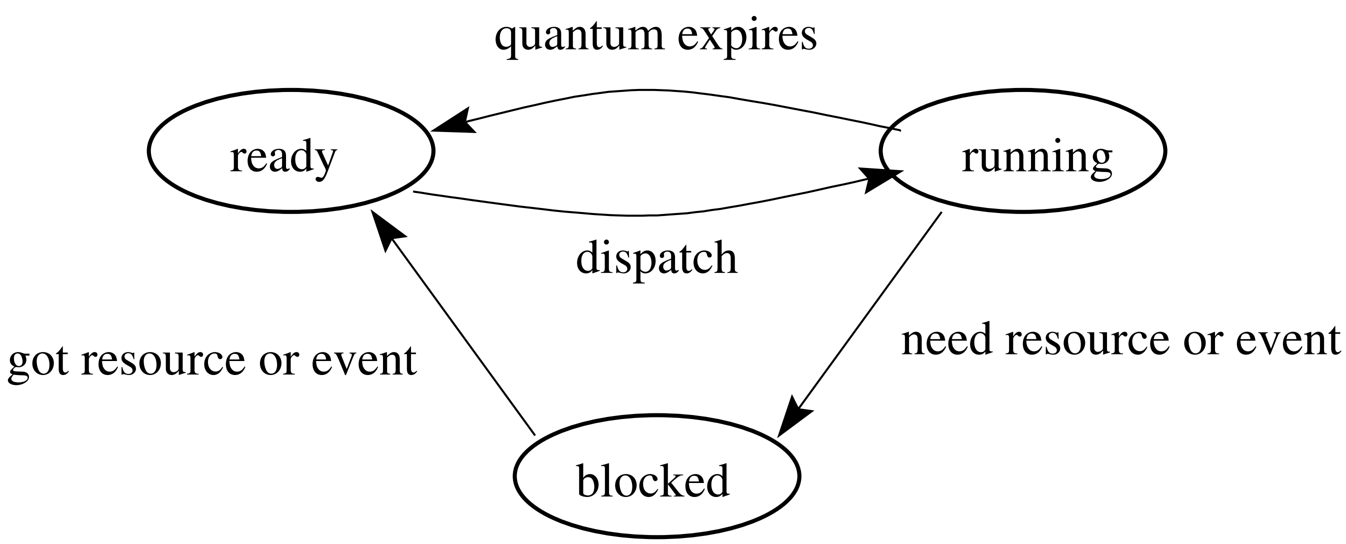
Multiprogramming makes it easier to keep the processor busy even though individual threads are not always ready.

Implementing Blocking

- The need for waiting normally arises during the execution of a system call by the thread, since programs use devices through the kernel (by making system calls).
- When the kernel recognizes that a thread faces a delay, it can *block* that thread. This means:
  - mark the thread as blocked, don't put it on the ready queue
  - choose a ready thread to run, and dispatch it
  - when the desired event occurs, put the blocked thread back on the ready queue so that it will (eventually) be chosen to run

Thread States

- a very simple process state transition diagram



- the states:

  running: currently executing

  ready: ready to execute

  blocked: waiting for something, so not ready to execute.

Summary of Hardware Features Used by the Kernel

Interrupts and Exceptions, such as timer interrupts, give the kernel the opportunity to regain control from user programs.

Memory management features, such as memory protection, allow the kernel to protect its address space from user programs.

Privileged execution mode allows the kernel to reserve critical machine functions (e.g, halt) for its own use.

Independent I/O devices allow the kernel to schedule other work while I/O operations are on-going.

User-Level Threads

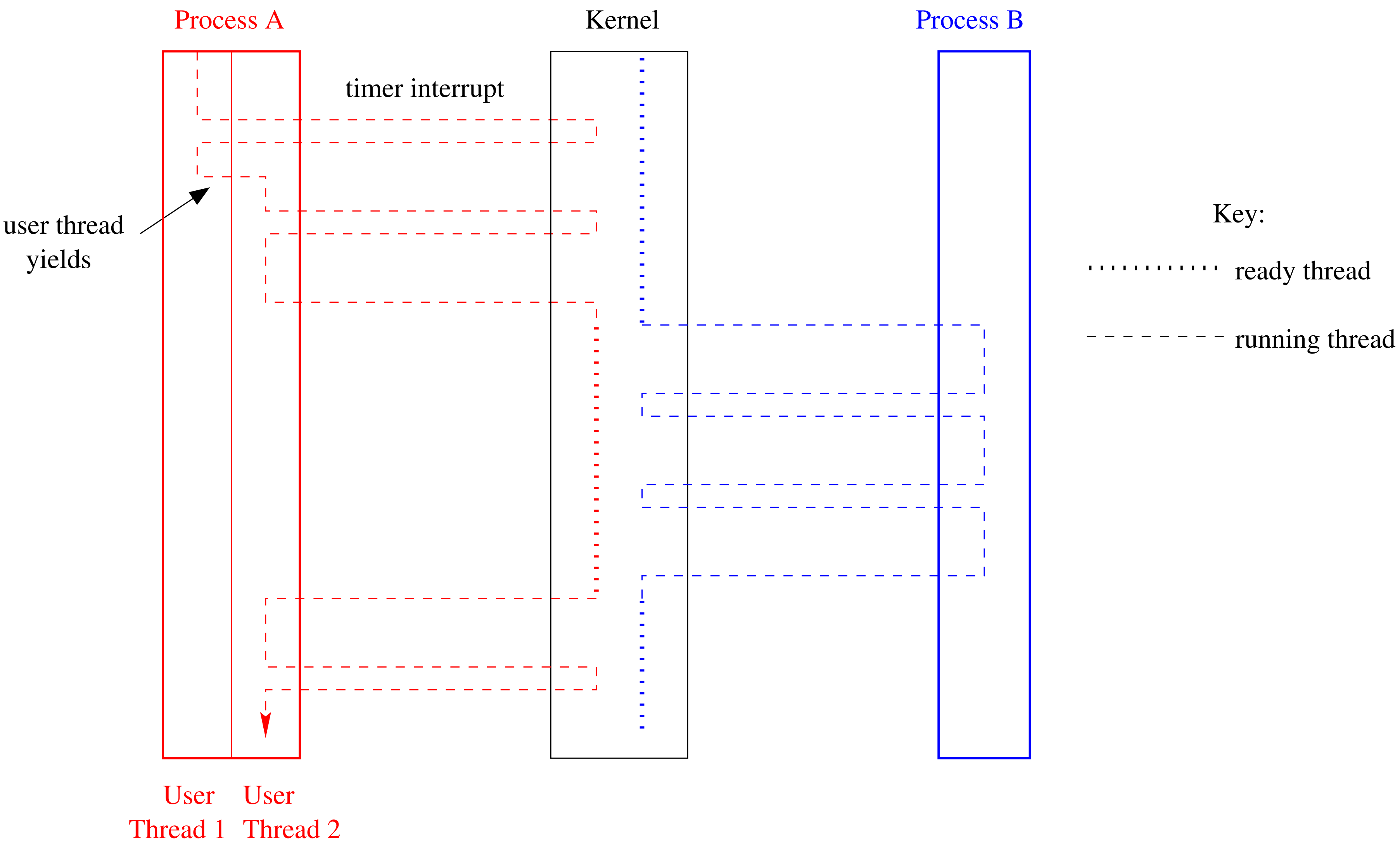
- It is possible to implement threading at the the user level.
- This means threads are implemented outside of the operating system, within a process.
- Call these *user-level threads* to distinguish them from *kernel threads*, which are those implemented by the operating system.
- A user-level thread library will include procedures for
  - creating threads
  - terminating threads
  - yielding (voluntarily giving up the processor)
  - synchronization

  In other words, similar operations to those provided by the operating system for kernel threads.

User-Level and Kernel Threads

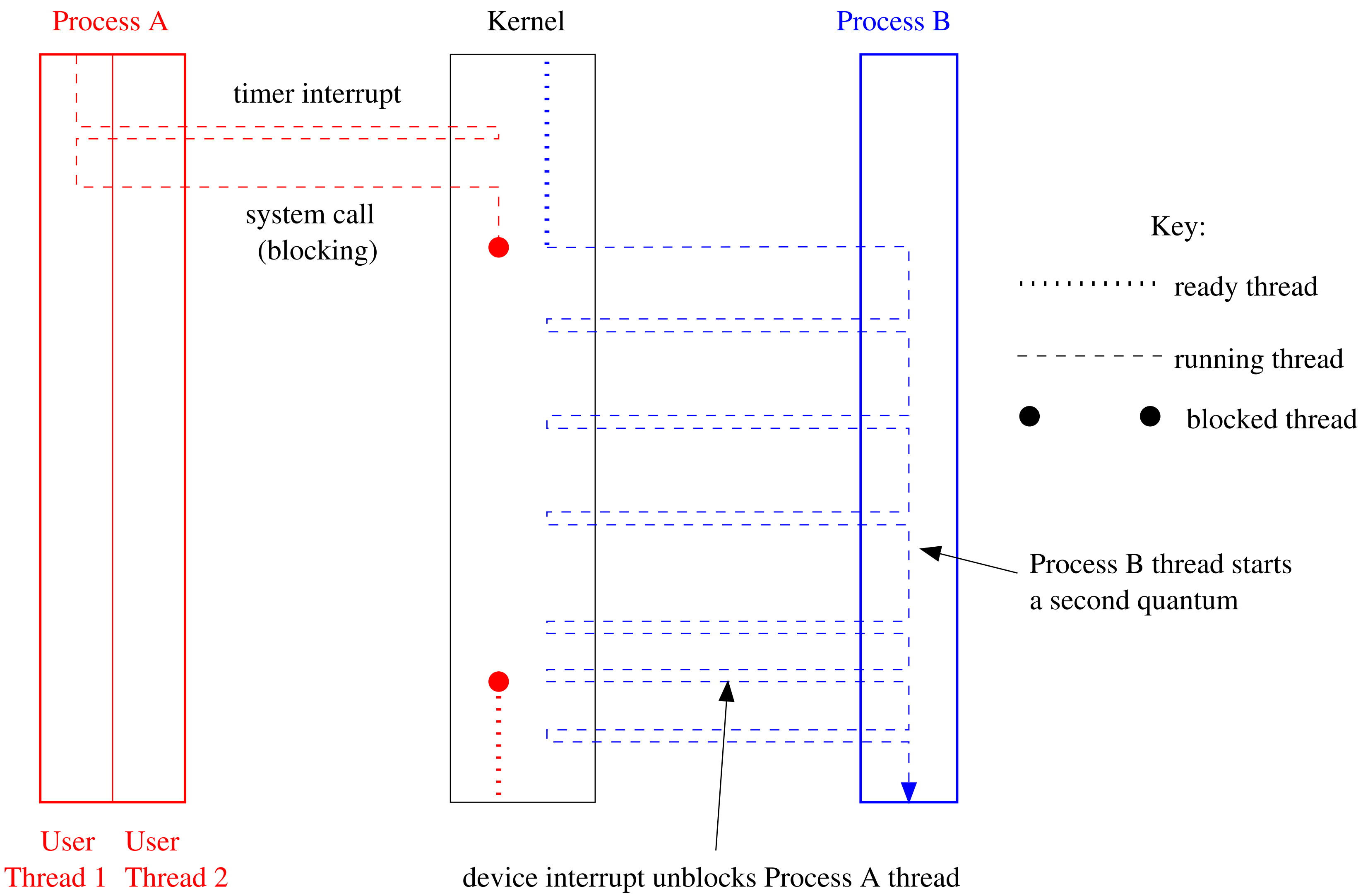
- There are two general ways to implement user-level threads
  1. Multiple user-level thread contexts in a process with one kernel thread. (N:1)
     - Kernel thread can "use" only one user-level thread context at a time.
     - Switching between user threads in the same process is non-preemptive.
     - Blocking system calls block the kernel thread, and hence all user threads in that process.
  2. Multiple user-level thread contexts in a process with multiple kernel threads. (N:M)
     - Each kernel thread "uses" one user-level thread context.
     - Switching between threads in the same process can be preemptive.
     - Process can make progress if at least one of its kernel threads is not blocked.

Two User Threads, One Kernel Thread (N:1 Example)



Process A has two user-level threads, but only one kernel thread.

User Thread Blocking (N:1 Example)



Once Process A's thread blocks, only Process B's thread can run.

Two User Threads, Two Kernel Threads (N:M Example)