What is a Process?

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

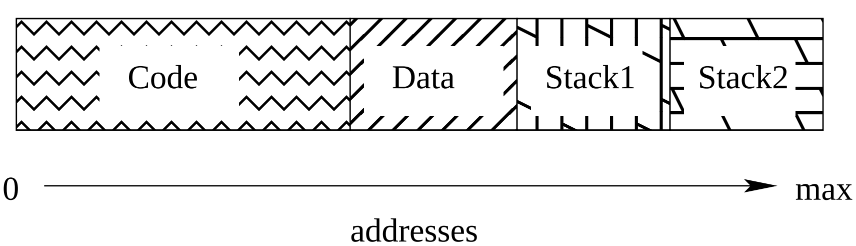
Answer 2: a process consists of

- 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.

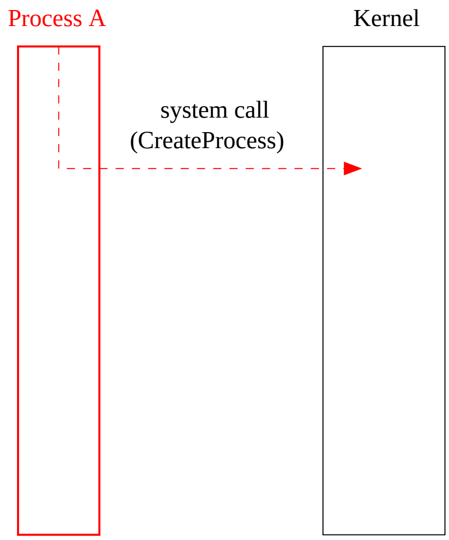
What is a Thread?

- 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

Implementation of Processes

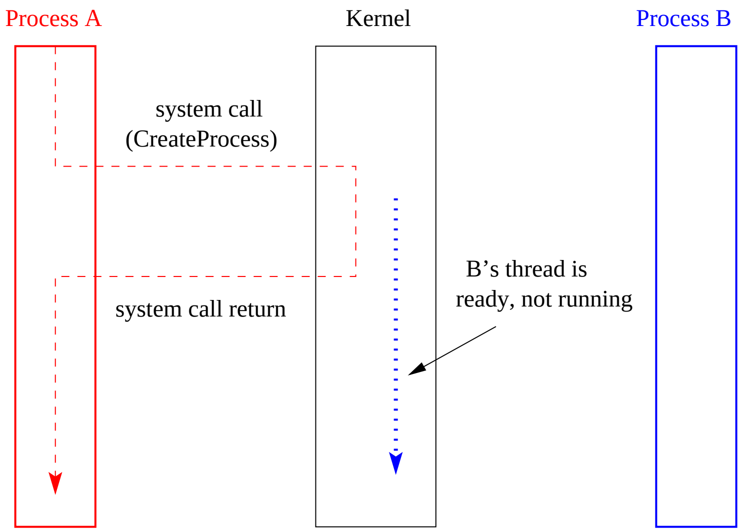
- The kernel maintains information about all of the processes in the system in a data structure often called the process table.
- Information about individual processes is stored in a structure that is sometimes called a *process control block (PCB)*. In practice, however, information about a process 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 . . ....

Process Creation Example (Part 1)



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

Process Creation Example (Part 2)

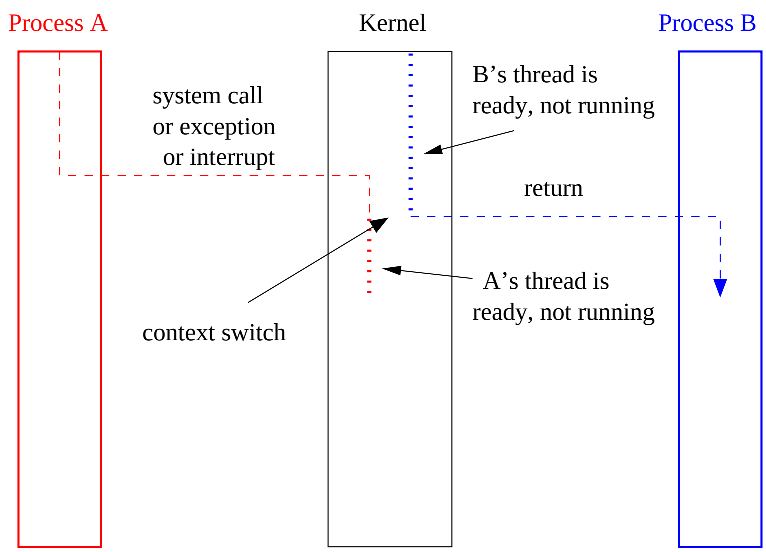


Kernel creates new process (Process B)

Multiprogramming

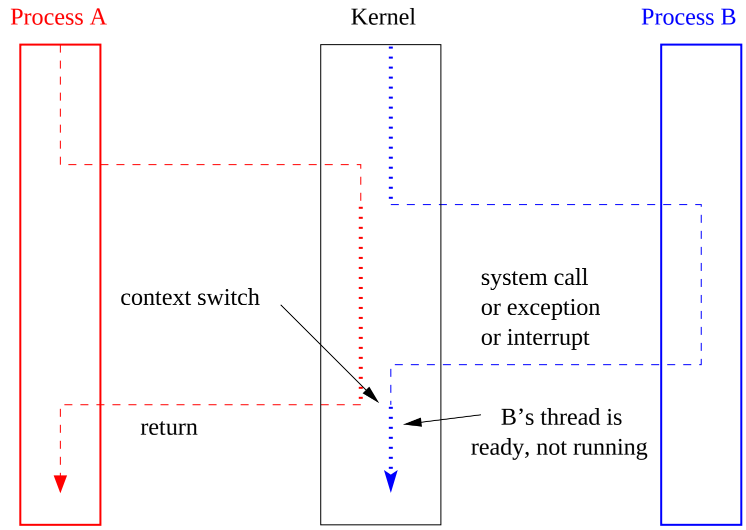
- multiprogramming means having multiple processes existing at the same time
- most modern, general purpose operating systems support multiprogramming
- all processes share the available hardware resources, with the sharing coordinated by the operating system:
  - Each process uses some of the available 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.

Timesharing Example (Part 1)



Kernel switches execution context to Process B.

Timesharing Example (Part 2)



Kernel switches execution context back to process A.

Process Interface

- A running program 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., `fork/exec/execv`

  Destruction: terminate a process, e.g., `exit`

  Synchronization: wait for some event, e.g., `wait/waitpid`

  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?

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.

Dispatching: Context Switching

```
mips_switch:
    /* a0/a1 points to old/new thread's struct pcb. */

   /* Allocate stack space for saving 11 registers. 11*4 = 44 */
   addi sp, sp, -44

   /* Save the registers */
   sw ra, 40(sp)
   sw gp, 36(sp)
   sw s8, 32(sp)
   sw s7, 28(sp)
   sw s6, 24(sp)
   sw s5, 20(sp)
   sw s4, 16(sp)
   sw s3, 12(sp)
   sw s2, 8(sp)
   sw s1, 4(sp)
   sw s0, 0(sp)

   /* Store the old stack pointer in the old pcb */
   sw sp, 0(a0)
```

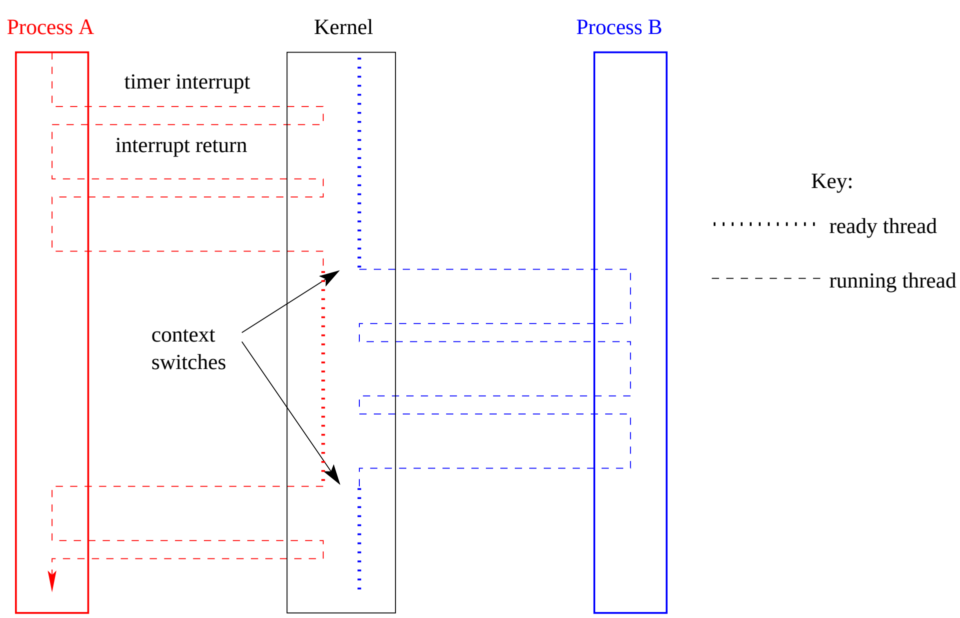

Dispatching: Context Switching

```
   /* Get the new stack pointer from the new pcb */
   lw sp, 0(a1)
   nop  /* delay slot for load */

   /* Now, restore the registers */
   lw s0, 0(sp)
   lw s1, 4(sp)
   lw s2, 8(sp)
   lw s3, 12(sp)
   lw s4, 16(sp)
   lw s5, 20(sp)
   lw s6, 24(sp)
   lw s7, 28(sp)
   lw s8, 32(sp)
   lw gp, 36(sp)
   lw ra, 40(sp)
   nop /* delay slot for load */

   j ra             /* and return. */
   addi sp, sp, 44 /* in delay slot */
   .end mips_switch
```

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.

Preemptive Multiprogramming 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.

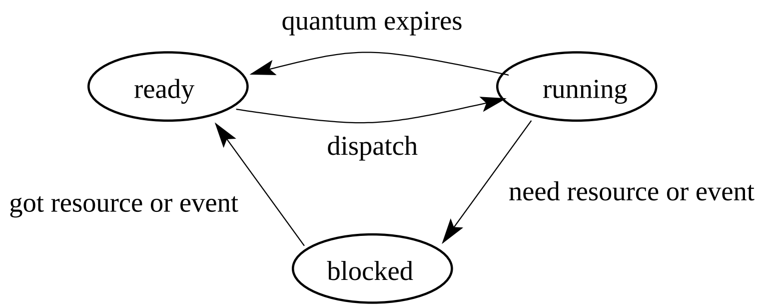
Multiprogramming makes it easier to keep the processor busy even though individual threads are not always ready.

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 thread state transition diagram



- the states:

  running: currently executing

  ready: ready to execute

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

User-Level Threads

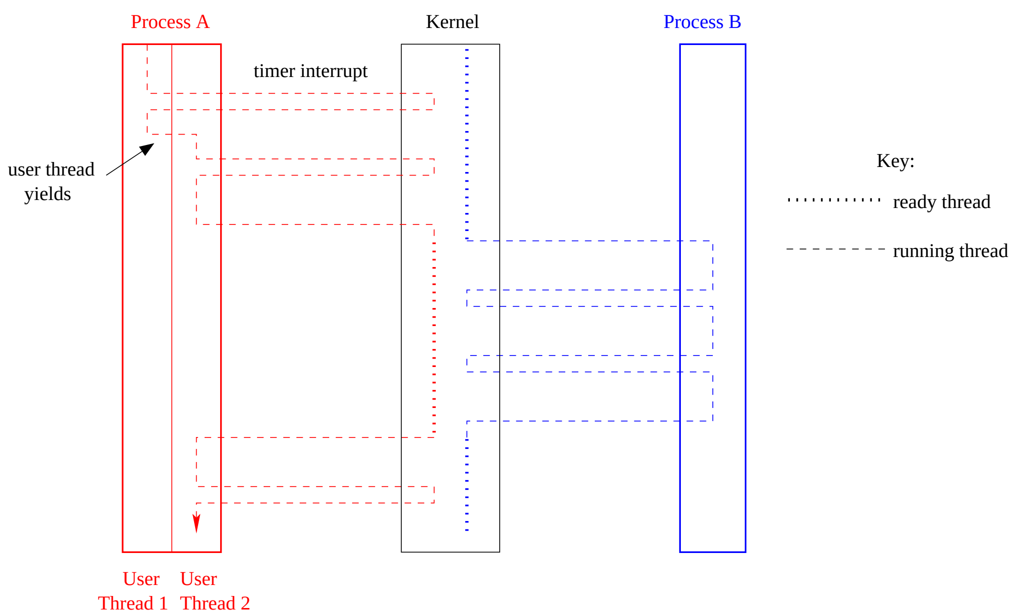
- It is possible to implement threading at the user level.
- This means threads are implemented outside of the kernel, within a process.
- Call these *user-level threads* to distinguish them from *kernel threads*, which are those implemented by the kernel.
- A user-level thread library will include procedures 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

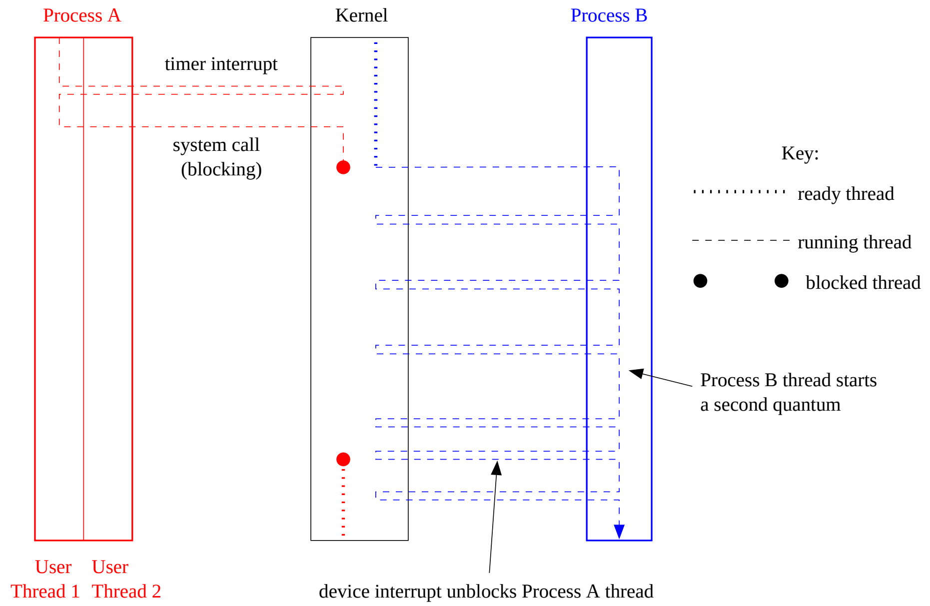
- There are two general ways to implement user-level threads
  1. Multiple user-level thread contexts in a process with one kernel thread. (N:1)
     - Kernel thread can "use" only one user-level thread context at a time.
     - Switching between user threads in the same process is typically non-preemptive.
     - Blocking system calls block the kernel thread, and hence all user threads in that process.
     - Can only use one CPU.
  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.
     - Can use multiple CPUs.

Two User Threads, One Kernel Thread (Part 1)



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

Two User Threads, One Kernel Thread (Part 2)



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

Two User Threads, Two Kernel Threads