# Recap

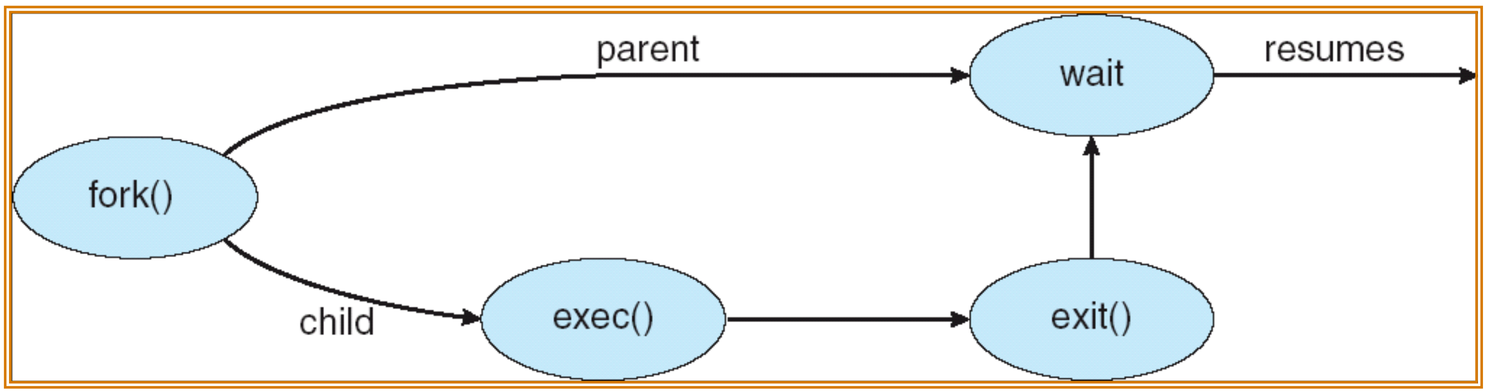
- Processes are programs in execution. Kernel represents a process using PCBs.
- Processes transition to various states (queues). Scheduling is the process of moving the processes in order to achieve a global goal (better interactive performance, throughput etc.)

# Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing policies between parent & child
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution model
  - Parent and children execute concurrently
  - Parent waits until children terminate

# Process Creation (Cont.)

- Address space of child
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program



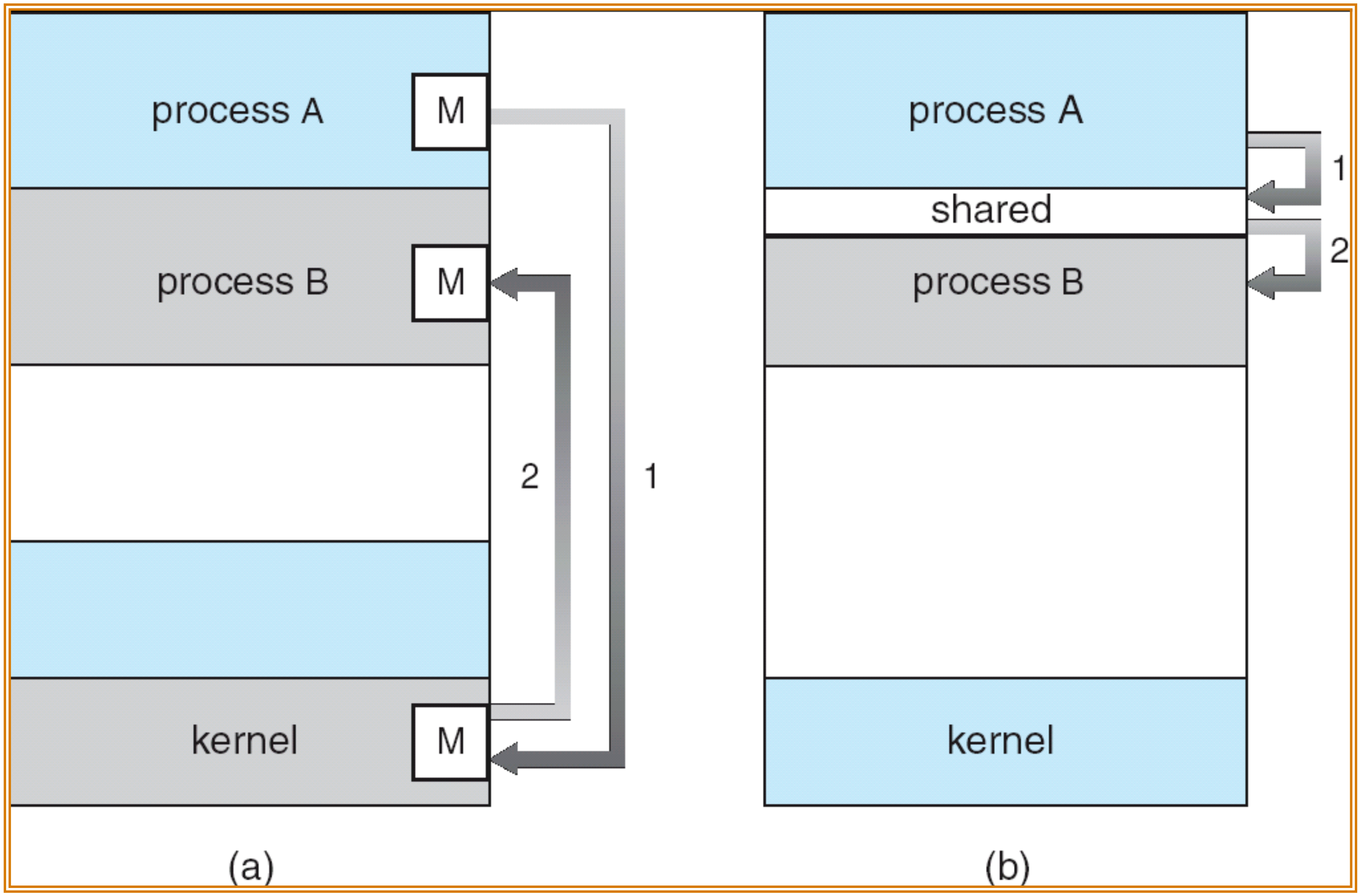
# Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*

# Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Producer-Consumer Problem
  - Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
    - unbounded-buffer places no practical limit on the size of the buffer
    - bounded-buffer assumes that there is a fixed buffer size

# Communications Models




messages

Shared memory

# Direct Communication

- Processes must name each other explicitly:
  - **send** (*P*, *msg*) – send a message to process P
  - **receive**(*Q*, *msg*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

# Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
  - **send**(*A, message*) – send a message to mailbox A
  - **receive**(*A, message*) – receive a message from mailbox A
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

# Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

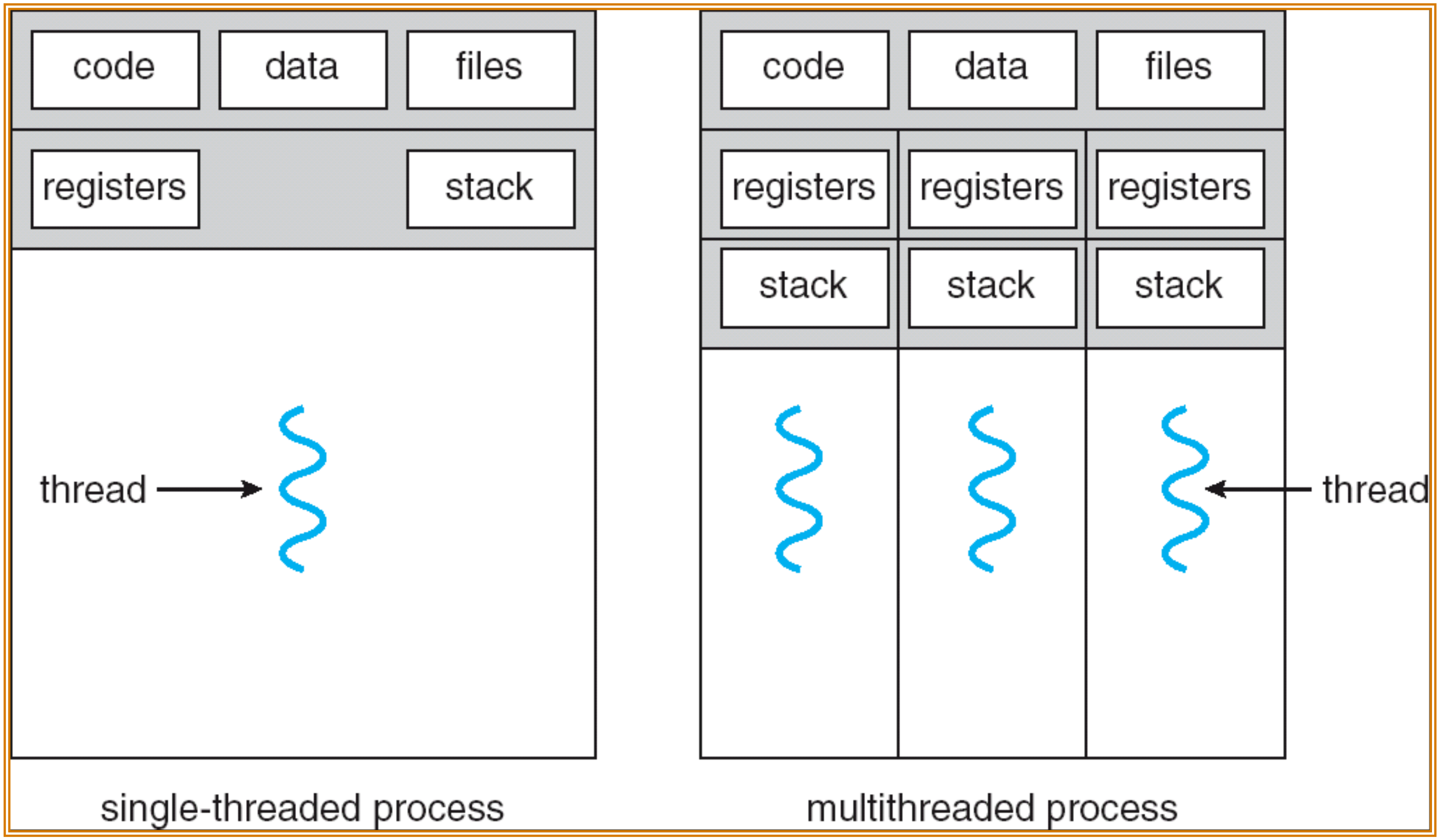
# Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

# Chapter 4: Threads

- Thread is the basic unit of CPU utilization. So far, our implicit assumption was that each process has a single thread of execution. However, each process can have multiple threads of execution, potentially working on more than one thing at the same time
- Threads in the same process share text, data, open files, signals and other resources. Each thread has its own execution context and stack.

# Single and Multithreaded Processes



# Benefits

- Responsiveness - Interactive applications can be performing two tasks at the same time (rendering, spell checking)
- Resource Sharing - Sharing resources between threads is easy (too easy?)
- Economy - Resource allocation between threads is fast (no protection issues)
- Utilization of MP Architectures - seamlessly assign multiple threads to multiple processors (if available). Future appears to be multi-core anyway.

# Thread types

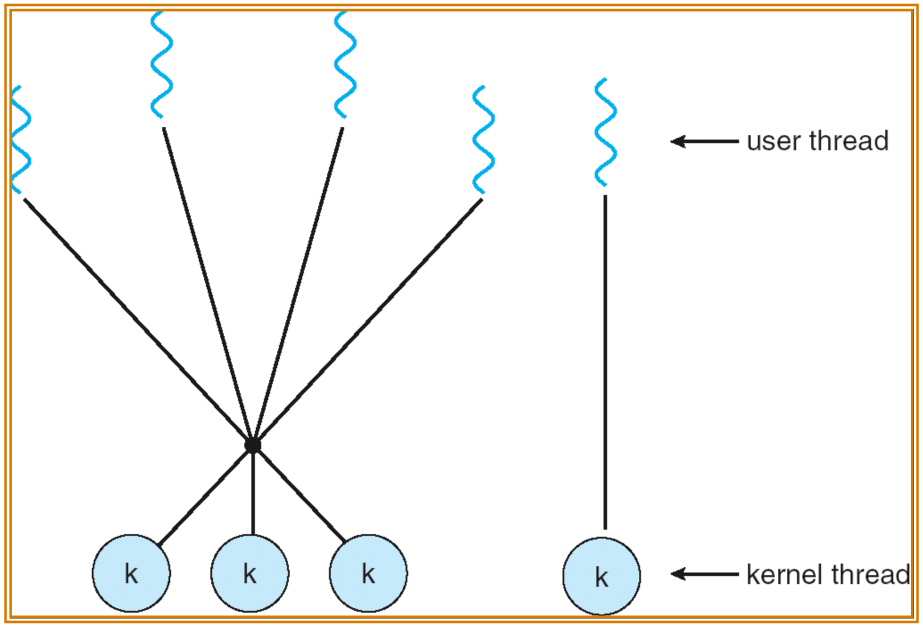
- User threads: thread management done by user-level threads library. Kernel does not know about these threads
  - Three primary thread libraries:
    - POSIX Pthreads
    - Win32 threads
    - Java threads
- Kernel threads: Supported by the Kernel and so more overhead than user threads
  - Examples: Windows XP/2000, Solaris, Linux, Mac OS X
- User threads map into kernel threads

# Multithreading Models

- Many-to-One: Many user-level threads mapped to single kernel thread
  - If a thread blocks inside kernel, all the other threads cannot run
  - Examples: Solaris Green Threads, GNU Pthreads
- One-to-One: Each user-level thread maps to kernel thread
- Many-to-Many: Allows many user level threads to be mapped to many kernel threads
  - Allows the operating system to create a sufficient number of kernel threads

# Two-level Model

- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



# Pthreads library

- Discuss the sample pthread program