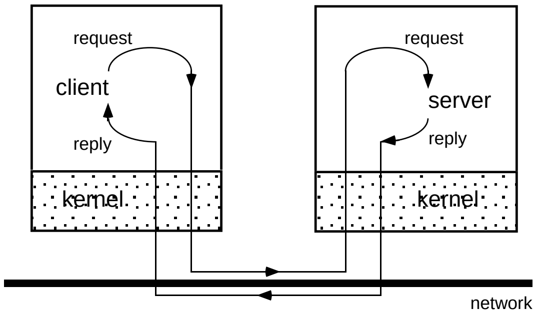
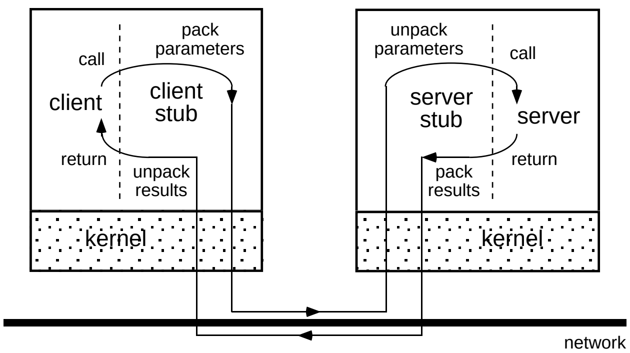
Client / Server Model using Message Passing and RPC (Review)

- Message Passing



- Remote Procedure Call (RPC)



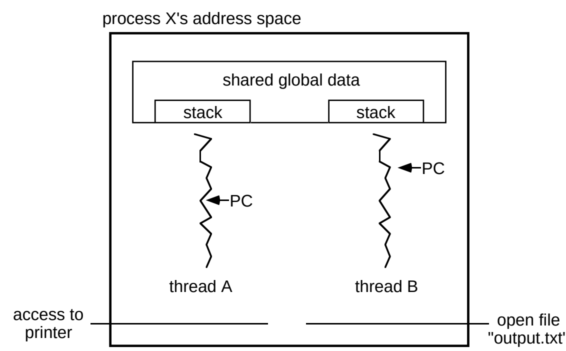
Conventional View of Processes

- A process can be viewed two ways:
  - A unit of **resource ownership**
    - A process has an address space, containing program code and data
    - A process may have open files, may be using an I/O device, etc.
  - A unit of **scheduling**
    - The CPU scheduler dispatches one process at a time onto the CPU
    - Associated with a process are values in the PC, SP, and other registers
- Insight (~1988) — these two are usually linked, but they don't have to be
- In many recent operating systems (UNIX, Windows NT), the two are independent:
  - Process = unit of resource ownership
  - Thread = unit of scheduling

Processes vs. Threads

- *Process* = unit of resource ownership
  - A process (sometimes called a *heavyweight process*) has:
    - Address space
    - Program code
    - Global variables, heap, stack
    - OS resources (files, I/O devices, etc.)
- *Thread* = unit of scheduling
  - A thread (sometimes called a *lightweight process*) is a single sequential execution stream within a process
  - A thread ***shares*** with other threads:
    - Address space, program code
    - Global variables, heap
    - OS resources (files, I/O devices)
  - A thread has its own:
    - Registers, Program Counter (PC)
    - Stack, Stack Pointer (SP)

Processes vs. Threads



- A thread is bound to a particular process
  - A process may contain multiple threads of control inside it
  - Threads can block, create children, etc.
- All of the threads in a process:
  - Share address space, program code, global variables, heap, and OS resources
  - Execute concurrently (has its own register, PC, SP, etc. values)

Why Threads?

- A process with multiple threads makes a great server (e.g., printer server):
  - Have one server process, many "worker" threads — if one thread blocks (e.g., on a read), others can still continue executing
  - Threads can share common data; don't need to use inter-process communication
  - Can take advantage of multiprocessors
- Threads are cheap!
  - Cheap to create — only need a stack and storage for registers
  - Use very little resources — don't need new address space, global data, program code, or OS resources
  - Context switches are fast — only have to save / restore PC, SP, and registers
- But… no protection between threads!

What Kinds of Programs Can Be Multithreaded?

- Good programs to multithread:
  - Server which needs to process multiple requests simultaneously
  - Programs with multiple independent tasks (debugger needs to run and monitor program, keep its GUI active, and display an interactive data inspector and dynamic call grapher)
  - Repetitive numerical tasks — break large problem, such as weather prediction, down into small pieces and assign each piece to a separate thread
- Programs difficult to multithread:
  - Programs that don't require any multiprocessing (99% of all programs)
  - Programs that require multiple processes (maybe one needs to run as root)

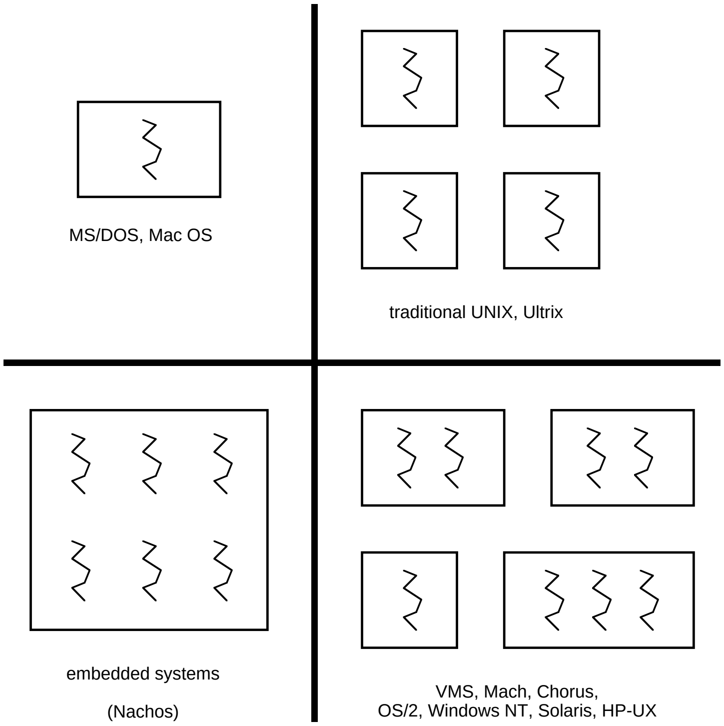
Using Threads in a Server

- Dispatcher-worker model
  - *Dispatcher* thread receives all requests, hands each to an idle *worker* thread, worker thread processes request
  - Worker threads are either created dynamically, or a fixed-size pool of workers is created when the server starts
- Team model
  - All threads are equals; each thread processes incoming requests on its own
  - Good for handling multiple types of requests within a single server
- Pipeline model
  - First thread partially processes request, then hands it off to second thread, which processes some more, then hands it off to third thread, etc.

The "Bank" Analogy

- Multiple tellers perform the same job — handling deposits, withdrawals, etc.
  - Customers wait in a queue for next available teller, go to whomever is free (one teller is the same as any other)
- Multiple officers perform other jobs — opening accounts, wiring money, etc.
- Bank has physical resources — desks, chairs, vault, teller stations, etc. — all tellers and officers share those resources
- If customer base increases, it's easy to add more tellers
  - If one teller gets tied up handling a difficult customer, other tellers can continue processing customers
  - It's much harder to build a new bank

Classifying Threaded Systems



User-Level Threads

- User-level threads = provide a library of functions to allow user processes to create and manage their own threads
  - ✔ Doesn't require modification to the OS
  - ✔ Simple representation — each thread is represented simply by a PC, registers, stack, and a small control block, all stored in the user process' address space
  - ✔ Simple management — creating a new thread, switching between threads, and synchronization between threads can all be done without intervention of the kernel
  - ✔ Fast — thread switching is not much more expensive than a procedure call
  - ✔ Flexible — CPU scheduling (among those threads) can be customized to suit the needs of the algorithm

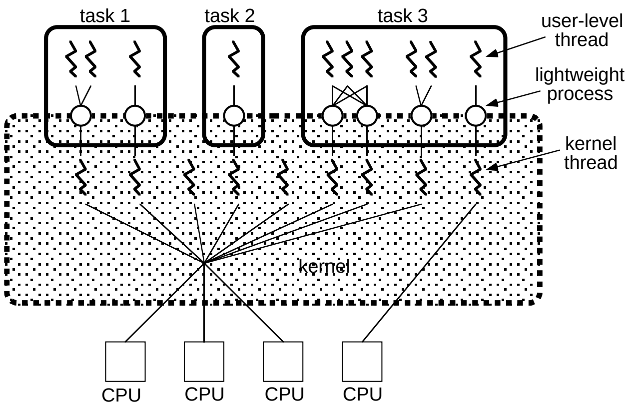
User-Level Threads (cont.)

- User-level threads = provide a library of functions to allow user processes to create and manage their own threads
  - ✘ Lack of coordination between threads and OS kernel
    - Process as a whole gets one time slice
    - Same time slice, whether process has 1 thread or 1000 threads
    - Also — up to each thread to relinquish control to other threads in that process
  - ✘ Requires non-blocking system calls (i.e., a multithreaded kernel)
    - Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the process
  - ✘ If one thread causes a page fault, the entire process blocks

Kernel-Level Threads

- Kernel-level threads = kernel provides system calls to create and manage threads
  - ✔ Kernel has full knowledge of all threads
    - Scheduler may choose to give a process with 10 threads more time than process with only 1 thread
  - ✔ Good for applications that frequently block (e.g., server processes with frequent interprocess communication)
  - ✘ Slow — thread operations are 100s of times slower than for user-level threads
  - ✘ Significant overhead and increased kernel complexity — kernel must manage and schedule threads as well as processes
    - Requires a full thread control block (TCB) for each thread

Two-Level Thread Model (Digital UNIX, Solaris, IRIX, HP-UX)



- User-level threads for user processes
  - "Lightweight process" (LWP) serves as a "virtual CPU" where user threads can run
- Kernel-level threads for use by kernel
  - One for each LWP
  - Others perform tasks not related to LWPs
- OS supports multiprocessor systems

Two-Level Thread Model (cont.)

- Process is called a "task", and contains user-level threads and LWPs
  - A set of user-level threads can be multiplexed over one or more LWPs
  - It's up to the process/task to schedule user-level threads onto LWPs
  - If a user-level thread blocks, the LWP and its associated kernel thread continue
- The OS only schedules kernel threads
  - If a kernel thread blocks, all its LWPs and user-level threads block
- A set of kernel-level threads may be multiplexed over a set of processors
  - Good for multiprocessors
  - Other kernel-level threads can be *pinned* to a specific processor