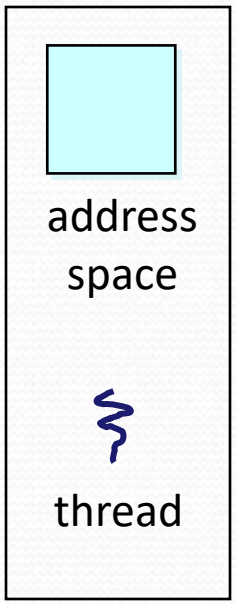
Lecture 4
Advanced Operating Systems
Threads

Overview of today's lecture

- The design space for threads
- Threads illustrated and view in an address space
- User level and kernel level thread implementations
- Problems and advantages of user level thread implementations
- Problems and advantages of kernel level thread implementations

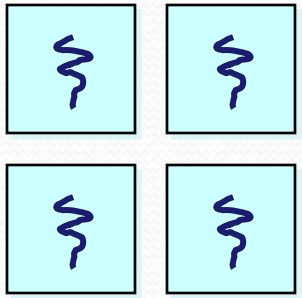
The design space



MS/DOS

one thread/process

one process

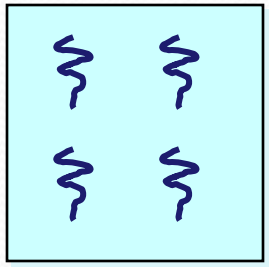


older UNIXes

one thread/process

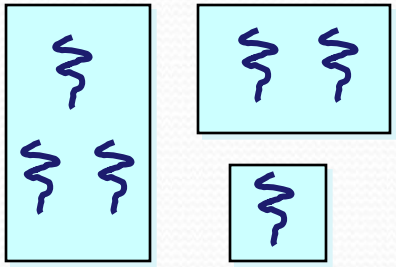
many processes

Java



many threads/process

one process

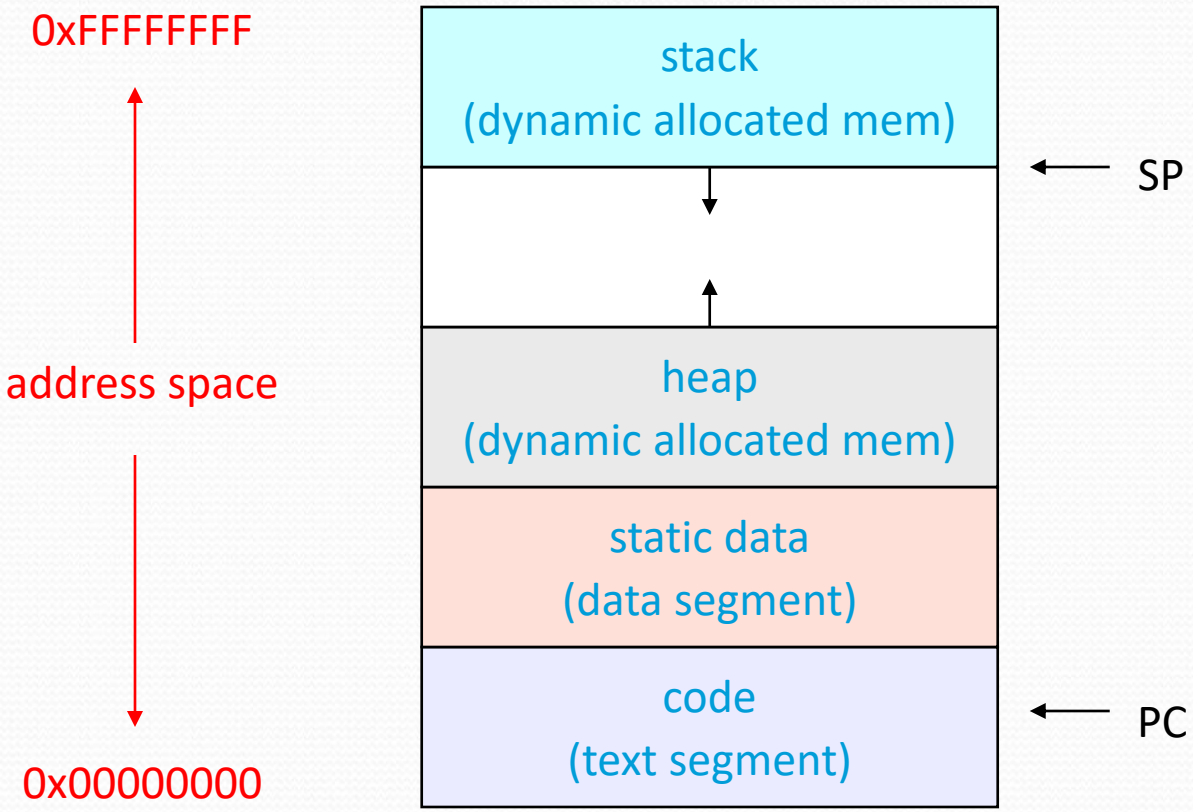


Mach, NT, Chorus, Linux, …

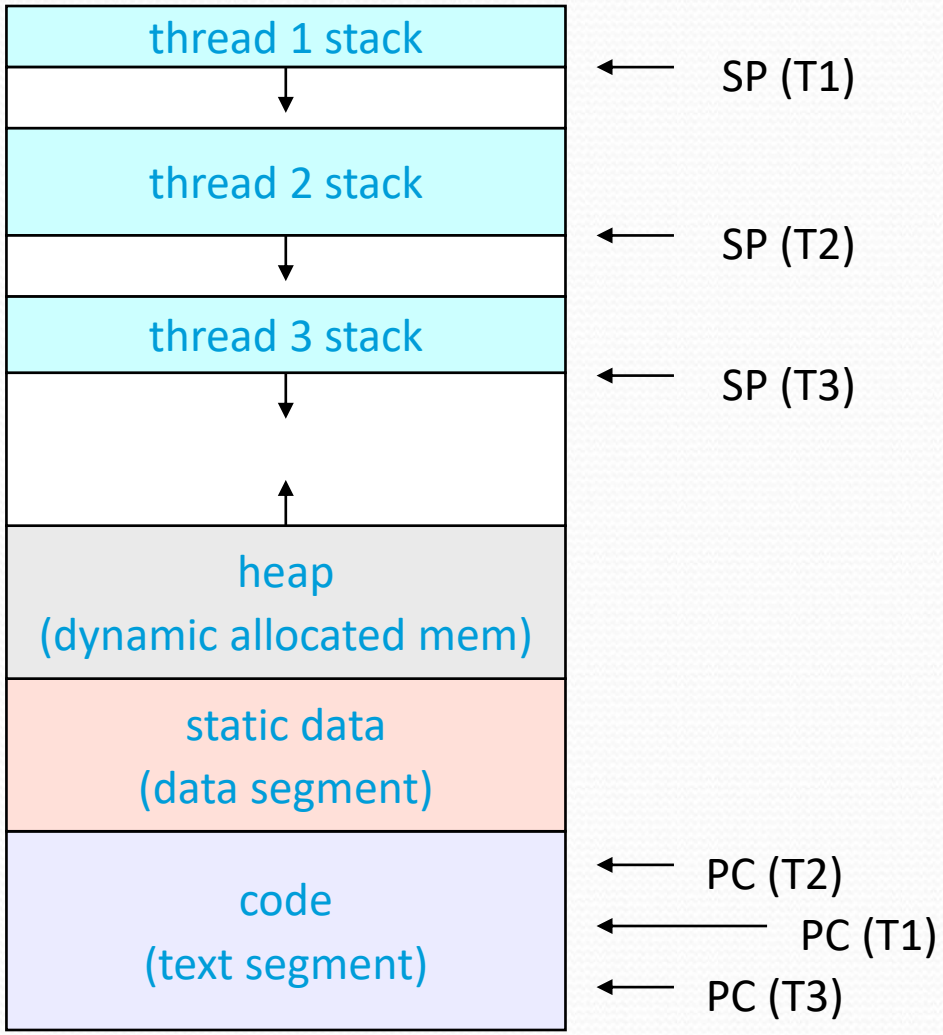
many threads/process

many processes

(old) Process address space



(new) Process address space with threads



Process/thread separation

- Concurrency (multithreading) is useful for:
  - handling concurrent events (e.g., web servers and clients)
  - building parallel programs (e.g., matrix multiply, ray tracing)
  - improving program structure (the Java argument)
- Multithreading is useful even on a uniprocessor
  - even though only one thread can run at a time
- Supporting multithreading – that is, separating the concept of a process (address space, files, etc.) from that of a minimal thread of control (execution state), is a big win
  - creating concurrency does not require creating new processes
  - "faster / better / cheaper"

Kernel threads

- OS manages threads *and* processes
  - all thread operations are implemented in the kernel
  - OS schedules all of the threads in a system
    - if one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
    - possible to overlap I/O and computation inside a process
- Kernel threads are cheaper than processes
  - less state to allocate and initialize
- But, they're still pretty expensive for fine-grained use (e.g., orders of magnitude more expensive than a procedure call)
  - thread operations are all system calls
    - context switch
    - argument checks
  - must maintain kernel state for each thread

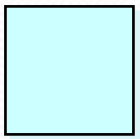
User-level threads

- To make threads cheap and fast, they need to be implemented at the user level
  - managed entirely by user-level library, e.g., **`libpthreads.a`**
- User-level threads are small and fast
  - each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB)
  - creating a thread, switching between threads, and synchronizing threads are done via procedure calls
    - no kernel involvement is necessary!
  - user-level thread operations can be 10-100x faster than kernel threads as a result

User-level thread implementation

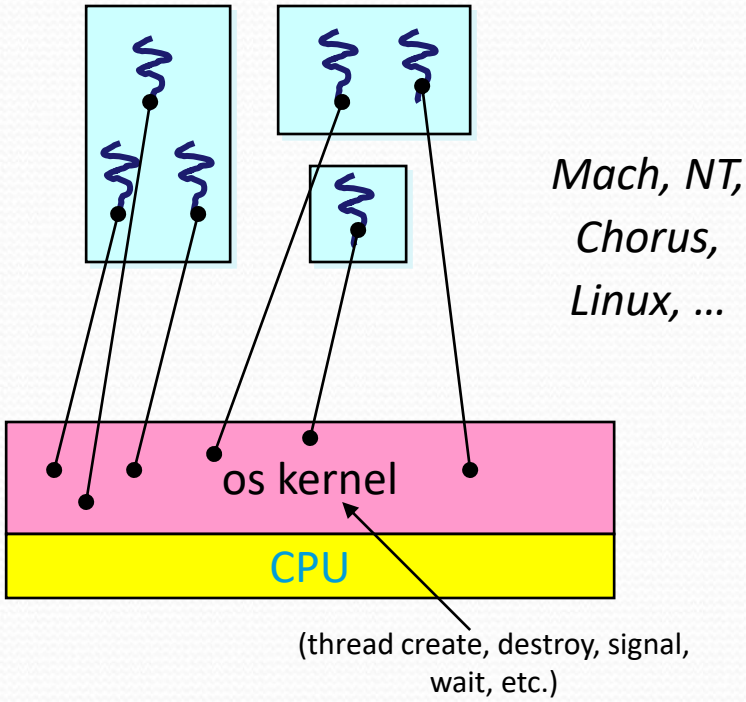
- The kernel believes the user-level process is just a normal process running code
  - But, this code includes the thread support library and its associated thread scheduler
- The thread scheduler determines when a thread runs
  - it uses queues to keep track of what threads are doing: run, ready, wait
    - just like the OS and processes
    - but, implemented at user-level as a library

Kernel threads



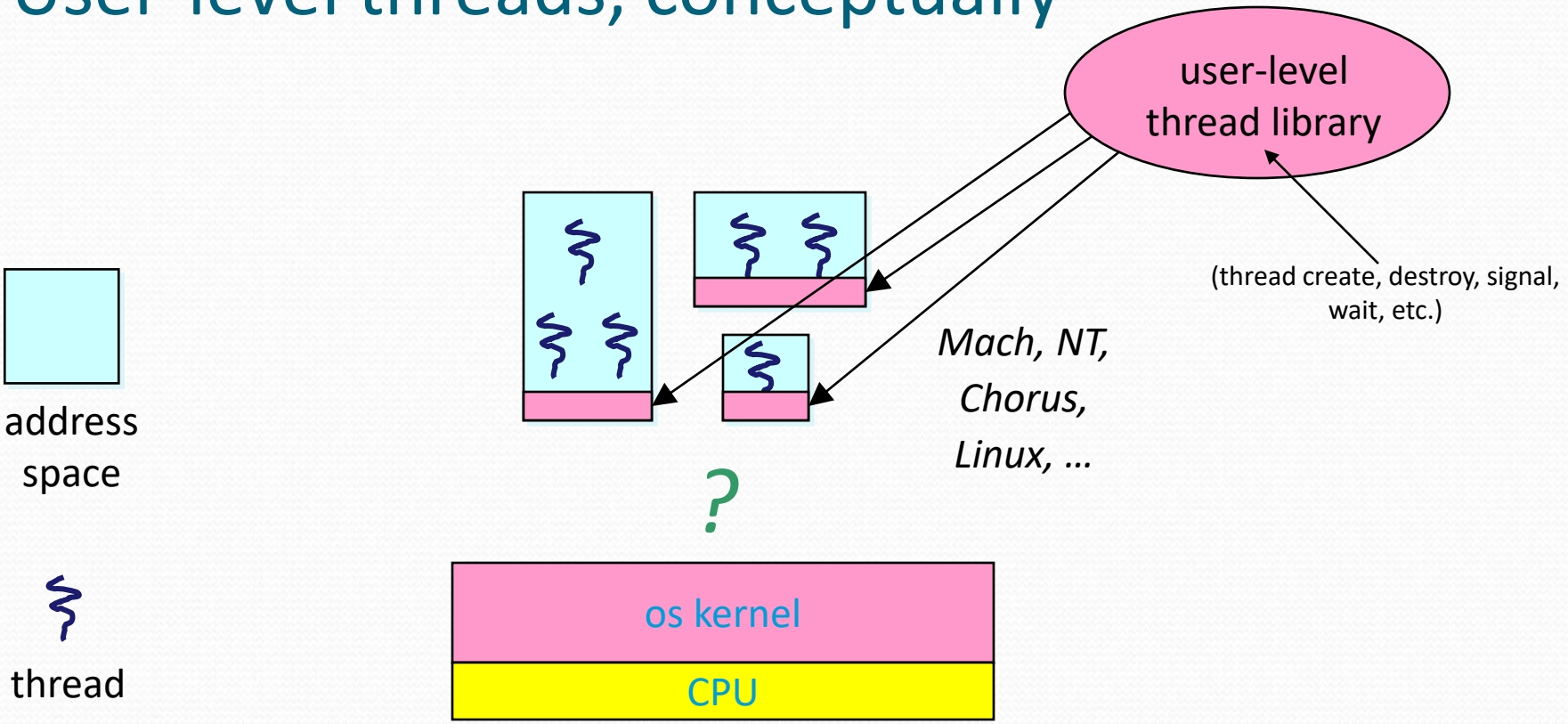
address space

thread

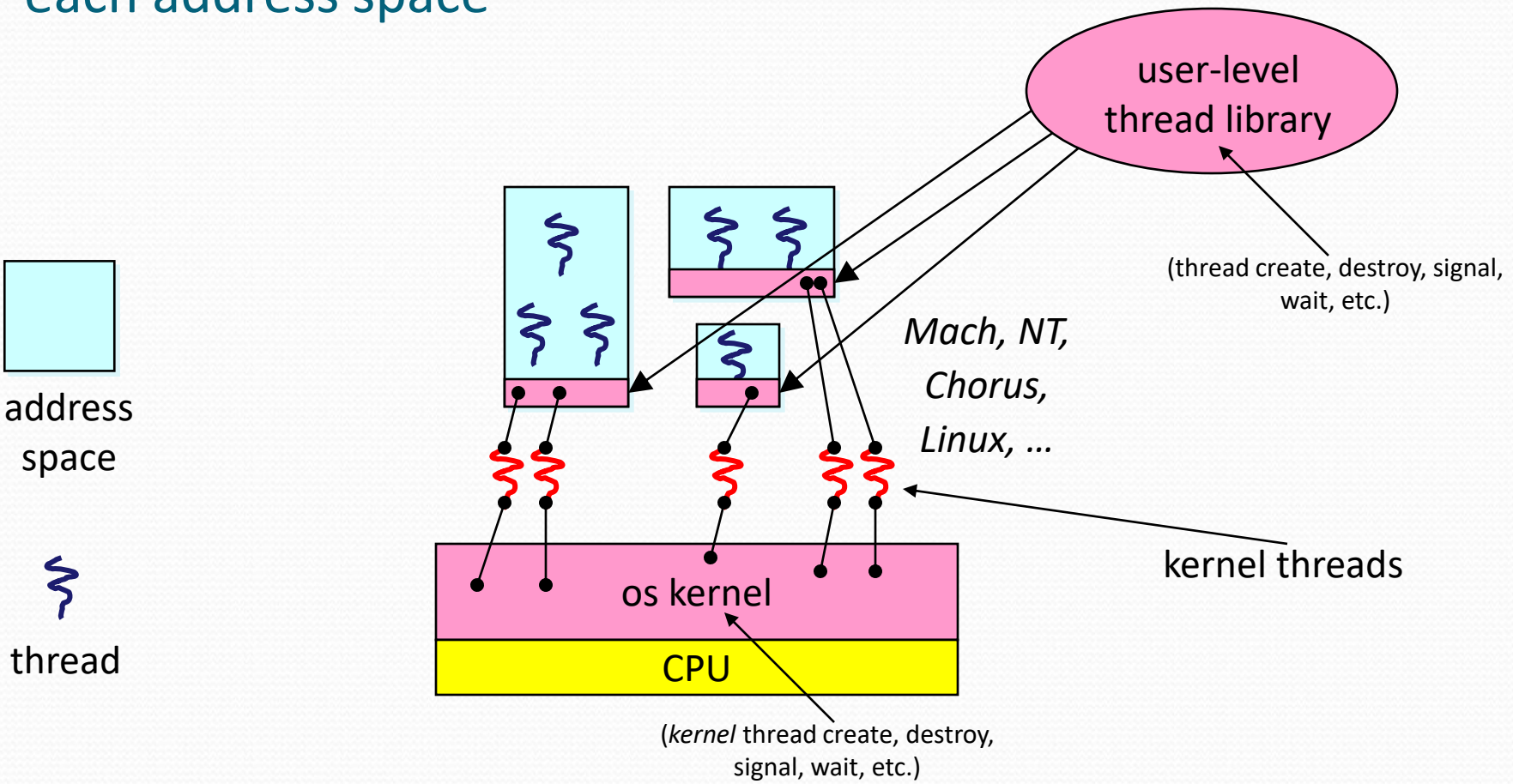


Mach, NT, Chorus, Linux, …

User-level threads, conceptually



Multiple kernel threads "powering" each address space



How to keep a user-level thread from hogging the CPU?

- Strategy 1: force everyone to cooperate
  - a thread willingly gives up the CPU by calling **`yield()`**
  - **`yield()`** calls into the scheduler, which context switches to another ready thread
  - what happens if a thread never calls **`yield()`**?

- Strategy 2: use preemption
  - scheduler requests that a timer interrupt be delivered by the OS periodically
    - usually delivered as a UNIX signal (`man signal`)
    - signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
  - at each timer interrupt, scheduler gains control and context switches as appropriate

Thread context switch

- Very simple for user-level threads:
  - save context of currently running thread
    - push machine state onto thread stack
  - restore context of the next thread
    - pop machine state from next thread's stack
  - return as the new thread
    - execution resumes at PC of next thread
- This is all done by assembly language
  - it works at the level of the procedure calling convention
    - thus, it cannot be implemented using procedure calls
    - e.g., a thread might be preempted (and then resumed) in the middle of a procedure call
    - C commands setjmp and longjmp are one way of doing it

What if a thread tries to do I/O?

- The kernel thread "powering" it is lost for the duration of the (synchronous) I/O operation!
- Could have one kernel thread "powering" each user-level thread
  - no real difference from kernel threads – "common case" operations (e.g., synchronization) would be quick
- Could have a limited-size "pool" of kernel threads "powering" all the user-level threads in the address space
  - the kernel will be scheduling these threads, obliviously to what's going on at user-level

What if the kernel preempts a thread holding a lock?

- Other threads will be unable to enter the critical section and will block (stall)
  - tradeoff, as with everything else
- Solving this requires coordination between the kernel and the user-level thread manager
  - "scheduler activations"
    - each process can request one or more kernel threads
      - process is given responsibility for mapping user-level threads onto kernel threads
      - kernel promises to notify user-level before it suspends or destroys a kernel thread

Pros and Cons of User Level threads

Pros

- Procedure invocation instead of system calls results in fast scheduling and thus much better performance
- Could run on exisiting OSes that don't support threads
- Customized scheduling. Useful in many cases e.g. in case of garbage collection
- Kernel space not required for thread specific data. Scale better for large number of threads

Pros and Cons of User Level threads

Cons

- Blocking system calls affect all threads
- Solution1: Make the system calls non-blocking. Is this a good solution?
- Solution2: Write Jacket or wrapper routines with select. What about this solution? Requires re-writing parts of system call library.
- Similar problem occurs with page faults. The whole process blocks
- Voluntary yield to the run-time system necessary for context switch
- Solution: Run time system may request a clock signal.

Pros and Cons of User Level threads

- Programs that use multi-threading need to make system calls quite often. If they don't then there is usually no need to be multi-threaded.

Pros and cons of kernel level threads

- Blocking system calls cause no problem
- Page faults can be handled by scheduling another thread from the same process (if one is ready)
- Cost of system call substantially greater
- Solution: thread recycling; don't destroy thread data structures when the thread is destroyed.

Task struct contents

- **Scheduling information:** Information needed by Linux to schedule processes. A process can be normal or real time and has a priority. Real-time processes are scheduled before normal processes, and within each category, relative priorities can be used. A counter keeps track of the amount of time a process is allowed to execute.
- **Identifiers:** Each process has a unique process identifier and also has user and group identifiers. A group identifier is used to assign resource access privileges to a group of processes.
- **Interprocess communication:** Linux supports the IPC mechanisms found in UNIX SVR4
- **Links:** Each process includes a link to its parent process, links to its siblings (processes with the same parent), and links to all of its children.
- **Times and timers:** Includes process creation time and the amount of processor time so far consumed by the process. A process may also have associated one or more interval timers. A process defines an interval timer by means of a system call; as a result a signal is sent to the process when the timer expires. A timer may be single use or periodic.

Task struct contents (cont'd)

- **File system:** Includes pointers to any files opened by this process, as well as pointers to the current and the root directories for this process.
- **Address space:** Defines the virtual address space assigned to this process.
- **Processor-specific context:** The registers and stack information that constitute the context of this process.
- **State:** The execution state of the process. These include:
  - **Running:** This state value corresponds to two states. A Running process is either executing or it is ready to execute.
  - **Interruptible:** This is a blocked state, in which the process is waiting for an event, such as the end of an I/O operation, the availability of a resource, or a signal from another process.
  - **Uninterruptible:** This is another blocked state. The difference between this and the Interruptible state is that in an uninterruptible state, a process is waiting directly on hardware conditions and therefore will not handle any signals.
  - **Stopped:** The process has been halted and can only resume by positive action from another process. For example, a process that is being debugged can be put into the Stopped state.
  - **Zombie:** The process has been terminated but, for some reason, still must have its task structure in the process table.

See /usr/include/linux/sched.h for process states in the Linux kernel