CS194-24
Advanced Operating Systems Structures and Implementation
Lecture 5

Processes/Threads

February 5th, 2014
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs194-24

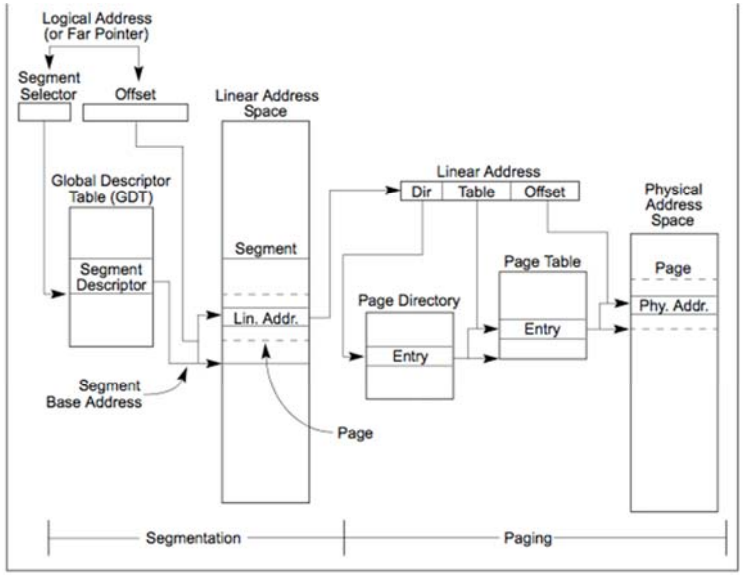
Goals for Today

- Processes
- Fork/Exec
- Multithreading/Posix support for threads
- Interprocess Communication

Interactive is important!
Ask Questions!

Note: Some slides and/or pictures in the following are adapted from slides ©2013

Recall: X86 Memory model with segmentation



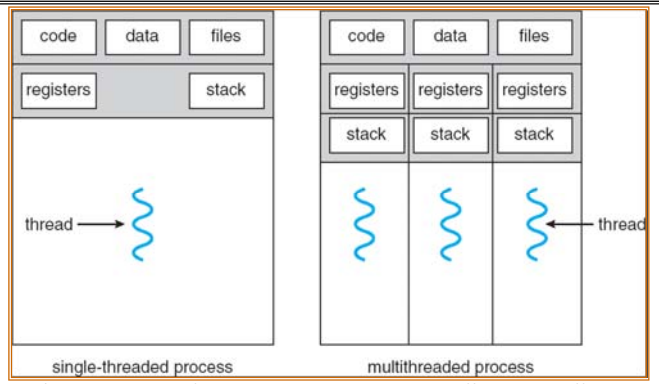
Recall: UNIX Process

- Process: *Operating system abstraction to represent what is needed to run a single program*
  - Originally: a single, sequential stream of execution in its *own* address space
  - Modern Process: multiple threads in same address space!
  - Different from a "program": a process is an *active* program and related resources
- Two parts:
  - Threads of execution (or just "threads")
    - » Code executed as one or more *sequential* stream of execution (threads)
    - » Each thread includes its own state of CPU registers
    - » Threads either multiplexed in software (OS) or hardware (simultaneous multithrading/hyperthreading)
  - Protected Resources:
    - » Main Memory State (contents of Address Space)
    - » I/O state (i.e. file descriptors)
- This is a virtual machine abstraction
  - Some might say that the only point of an OS is to support a clean Process abstraction

Recall: Modern "Lightweight" Process with Threads

- Thread: *a sequential execution stream within process* (Sometimes called a "Lightweight process")
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: *a single program made up of a number of different concurrent activities*
  - Sometimes called multitasking, as in Ada…
- Why separate the concept of a thread from that of a process?
  - Discuss the "thread" part of a process (concurrency)
  - Separate from the "address space" (Protection)
  - Heavyweight Process ≡ Process with one thread
- Linux confuses this model a bit:
  - Processes and Threads are "the same"
  - Really means: Threads are managed separately and can share a variety of resources (such as address spaces)
  - Threads related to one another in fashion similar to Processes with Threads within

Single and Multithreaded Processes

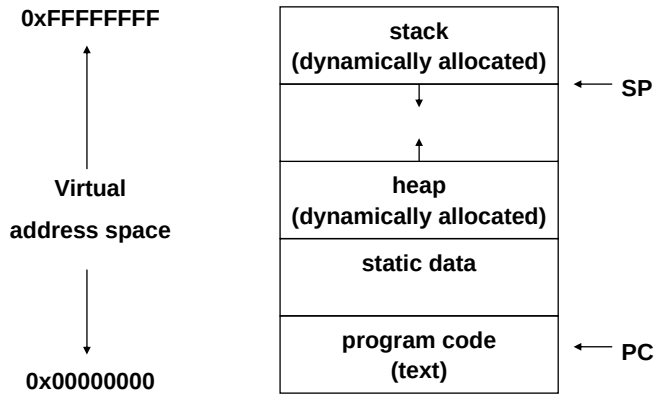


- Threads encapsulate concurrency: "Active" component
  - How do thread stacks stay separate from one another?
  - They may not!
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system

Process comprises

- What is in a process?
  - an address space – usually protected and virtual – mapped into memory
  - the code for the running program
  - the data for the running program
  - an execution stack and stack pointer (SP); also heap
  - the program counter (PC)
  - a set of processor registers – general purpose and status
  - a set of system resources
    - » files, network connections, pipes, …
    - » privileges, (human) user association, …
    - » Personalities (linux)
  - …

Processes – Address Space

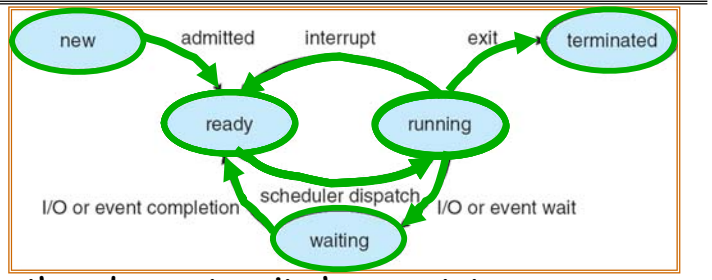


See also Silbershatz, figure 3.1

Process – starting and ending

- Processes are created …
  - When the system boots
  - By the actions of another process (more later)
  - By the actions of a user
  - By the actions of a batch manager
- Processes terminate …
  - Normally – exit
  - Voluntarily on an error
  - Involuntarily on an error
  - Terminated (killed) by the actions a user or a process

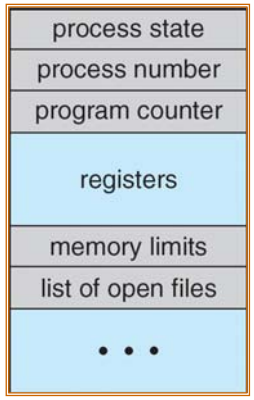
Lifecycle of a Process (or Thread)



- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
    Can be *Interruptible* or *Non-Interruptible*
  - terminated: The thread has finished execution
    Stays as *Zombie* until relays result to parent
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their state

How do we multiplex processes?

- The current state of process held in a process control block (PCB):
  - This is a "snapshot" of the execution and protection environment
  - Only one PCB active at a time
- Give out CPU time to different processes (Scheduling):
  - Only one process "running" at a time
  - Give more time to important processes
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
  - Sample mechanisms:
    - » Memory Mapping: Give each process their own address space
    - » Kernel/User duality: Arbitrary multiplexing of I/O through system calls



Process Control Block

Preview: System-Level Control of x86

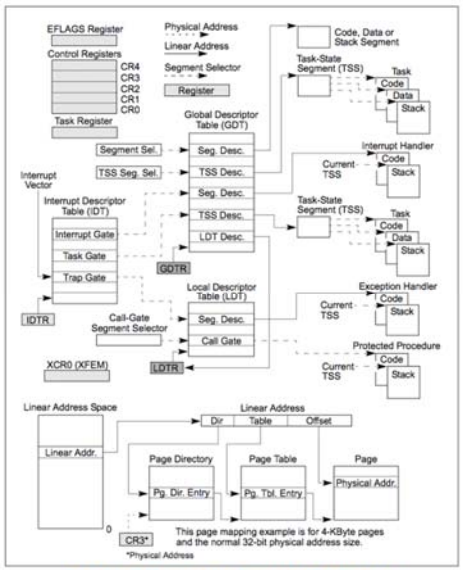


Figure 2-1. IA-32 System-Level Registers and Data Structures

- Full support for Process Abstraction involves a lot of system-level state
  - This is state that can only be accessed in kernel mode!
  - We will be talking about a number of these pieces as we go through the term…
- There is a tradeoff between amount of system state and cost of switching from thread to thread!

Processes in the OS – PCB

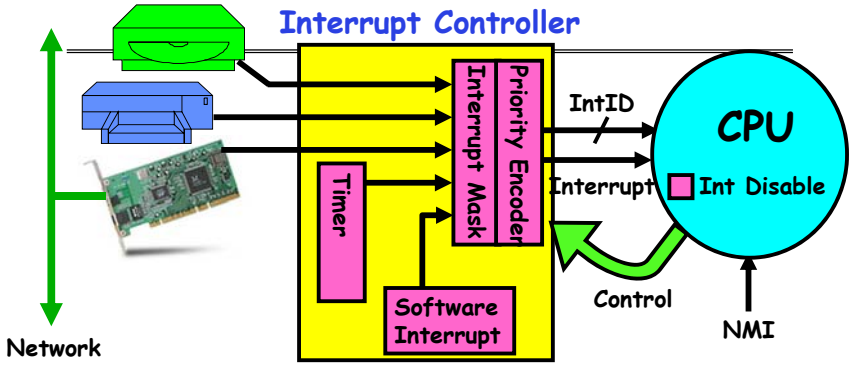
- Typical PCB contains:
  - execution state
  - PC, SP & processor registers – stored when process is not in *running* state
  - memory management info
  - Privileges and owner info
  - scheduling priority
  - resource info
  - accounting info
- Linux specific: PCB ⇒ task_struct
  - Linux generically calls threads of control "tasks"
  - The current macro is global and points at the current process/task:

```
struct task_struct *my_structure = current;
struct task_struct *my_parent = current->parent;
```


Administrivia

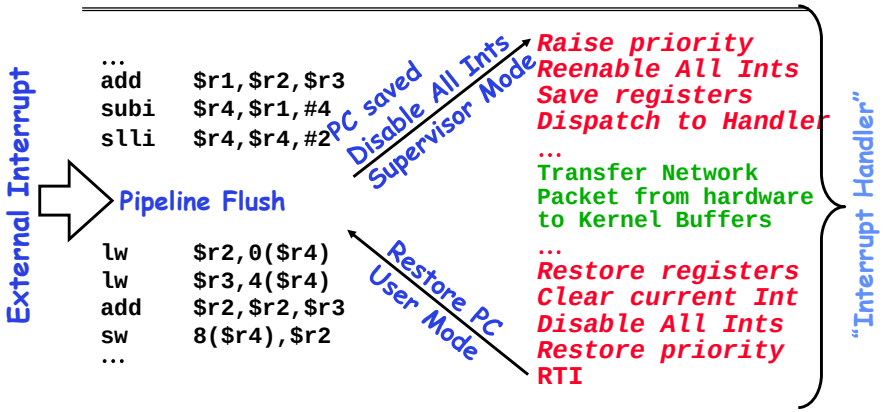
- Moved deadline for Lab 0.5 ⇒ Tomorrow@9pm
  - Lab 1 not quite ready yet – will post soon
- Still have one group of 3 and four(!) groups of 5
  - Isn't there anyone who can switch?
  - It is typically easier to work in groups of 4…!
  - Or, do we need to randomly pick someone???
- I've started posting video links on lecture page
  - Top link is for all of them, each lecture has individual link underneath slides

Interrupt Controller



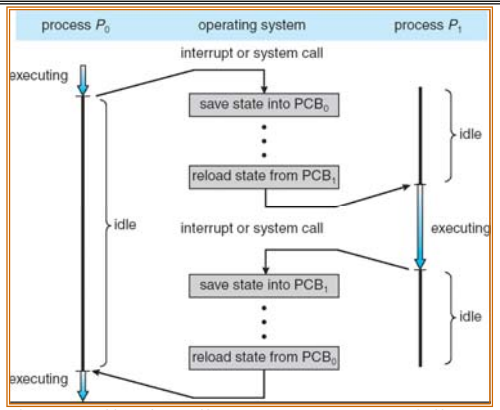
- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can't be disabled

Example: Network Interrupt



- Disable/Enable All Ints ⇒ Internal CPU disable bit
  - RTI reenables interrupts, returns to user mode
- Raise/lower priority: change interrupt mask
- Software interrupts can be provided entirely in software at priority switching boundaries

CPU Switch From Process to Process

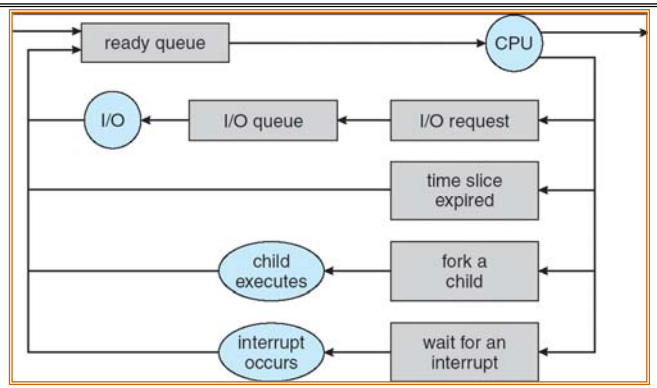


- This is also called a "context switch"
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but… contention for resources instead

Processes – State Queues

- The OS maintains a collection of *process state* queues
  - typically one queue for each state – e.g., ready, waiting, …
  - each PCB is put onto a queue according to its current state
  - as a process changes state, its PCB is unlinked from one queue, and linked to another
- Process state and the queues change in response to events – interrupts, traps

Process Scheduling



- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (few weeks from now)

Processes – Privileges

- Users are given privileges by the system administrator
- Privileges determine what *rights* a user has for an *object*.
  - Unix/Linux – Read|Write|eXecute by user, group and "other" (i.e., "world")
  - WinNT – *A*ccess *C*ontrol *L*ist
- Processes "inherit" privileges from user

Linux Kernel Implementation

- Kernel may execute in either *Process context* vs. *Interrupt context*
- In *Process context*, kernel has access to
  - Virtual memory, files, other process resources
  - May sleep, take page faults, etc., on behalf of process
    - » Can use functions that might sleep (e.g.`copy_from_user()`, or `kmalloc()` with GPP_USER or GPP_KERNEL flags)
  - The "`current`" macro is available.
- In *Interrupt context*, no assumption about what process was executing (if any)
  - No access to virtual memory, files, resources
  - May not sleep, take page faults, etc!
    - » Be careful what you call!
  - The "`current`" macro is not available.

System Calls for Creating Processes

- Begins life with the fork() system call
  - Creates new process by duplicating an existing one
  - Duplicate: memory, file descriptors, etc
  - The process that calls fork() is the parent, wheres the new process is the child.
- After creation, the parent resumes execution and the child starts execution at same place:
  - Where the call to fork() returns
  - i.e. fork() returns from the kernel twice!
- exit() system call:
  - Terminates the process and frees all its resources
  - A parent process can inquire about status of a terminated child via one of the wait system calls
  - Exiting process placed into special zombie state until parent calls wait or waitpid()

Process Creation – Unix & Linux

- Create a new (child) process – `fork();`
  - Allocates new PCB
  - Clones the calling process (almost exactly)
    - » Copy of parent process address space
    - » Copies resources in kernel (e.g. files)
  - Places new PCB on *Ready queue*
  - Return from `fork()` call
    - » 0 for child
    - » child PID for parent

Example of *fork( )*

```
int main(int argc, char **argv)
{
  char *name = argv[0];
  int child_pid = fork();
  if (child_pid == 0) {
    printf("Child of %s sees PID of %d\n",
           name, child_pid);
    return 0;

  } else {
    printf("I am the parent %s. My child is %d\n",
              name, child_pid);
    return 0;
  }
}
```

```
% ./forktest
Child of forktest sees PID of 0
I am the parent forktest. My child is 486
```

A (more compelling?) example of *fork( ): Web Server*

Serial Version

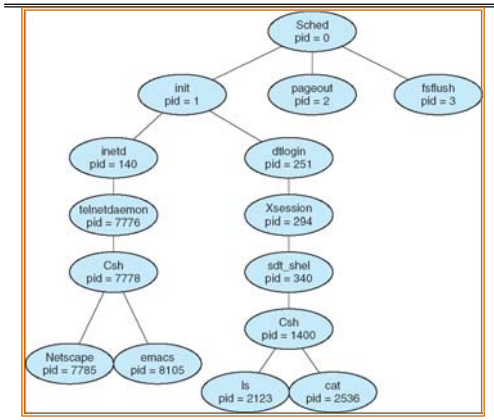
```
int main() {
  int listen_fd = listen_for_clients();
  while (1) {
    int client_fd = accept(listen_fd);
    handle_client_request(client_fd);
    close(client_fd);
  }
}
```

Process Per Request

```
int main() {
  int listen_fd = listen_for_clients();
  while (1) {
    int client_fd = accept(listen_fd);
    if (fork() == 0) {
      handle_client_request(client_fd);
      close(client_fd);    // Close FD in child when done
      exit(0);
    } else {
      close(client_fd);    // Close FD in parent

      // Let exited children rest in peace!
      while (waitpid(-1,&status,WNOHANG) > 0);
    }
}
```


Parent-Child relationship



Typical process tree for Solaris system

- Every Process has a parentage
  - A "parent" is a thread that creates another thread
  - A child of a parent was created by that parent

Starting New Programs

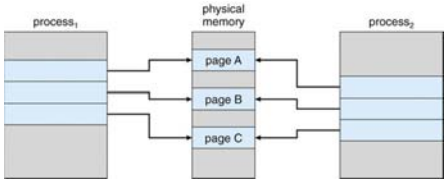
- Exec System Call:
  - `int exec (char *prog, char **argv)`
  - Check privileges and file type
  - Loads program at path *prog* into address space
    - » Replacing previous contents!
    - » Execution starts at `main()`
  - Initializes context – e.g. passes arguments
    - » `*argv`
  - Place PCB on *ready queue*
  - Preserves, pipes, open files, privileges, etc.
- Running a new program in Unix/Linux
  - fork() followed by exec()
  - Creates a new process as clone of previous one
  - First thing that clone does is to replace itself with new program

Fork + Exec – shell-like

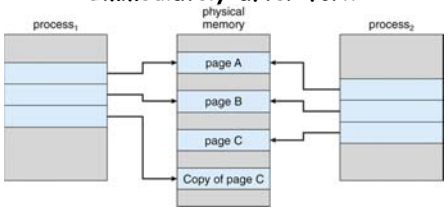
```
int main(int argc, char **argv)
{ char *argvNew[5];
  int pid;
  if ((pid = fork()) < 0) {
        printf( "Fork error\n");
        exit(1);
  } else if (pid == 0) { /* child process */
        argvNew[0] = "/bin/ls";
        argvNew[1] = "-l";
        argvNew[2] = NULL;
        if (execve(argvNew[0], argvNew, environ) < 0) {
           printf( "Execve error\n");
           exit(1);
        }
  } else { /* parent */
        wait(pid); /* wait for the child to finish */
  }
}
```

How does fork() actually work in Linux?

- Semantics are that fork() duplicates a lot from the parent:
  - Memory space, File Descriptors, Security Context
- How to make this cheap?
  - Copy on Write!
  - Instead of copying memory, copy page-tables
- Another option: vfork()
  - Same effect as fork(), but page tables not copied
  - Instead, child executes in parent's address space Parent blocked until child calls "exec()" or exits
  - Child not allowed to write to the address space



Immediately after fork



After Process-1 writes Page C

Clone system call in Linux

- What about other shared state?
  - Clone() system call!
  - Can select exactly what state to copy, what to create
  - Difference between creating process and thread in Linux is just different arguments to clone!
- Some interesting clone flags:
  - CLONE_FILES: Parent and Child share open files
  - CLONE_FS: Parent and child share filesystem info
  - CLONE_PARENT: Child has same parent as parent
  - CLONE_THREAD: Parent and child in same thread grp
  - CLONE_VFORK: do a vfork()
  - CLONE_VM: Parent and Child share addess space
  - CLONE_SIGHAND: Parent and Child share signal handlers and blocked signals
- So – implementations of calls:
  - Fork(): clone(SIGCHLD)
  - Vfork(): clone(CLONE_VFORK | CLONE_VM | SIGCHLD)
  - Thread creation: clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND)

Waiting for a Process

- Multiple variations of *wait* function
  - » Including non-blocking *wait* functions
- Waits until child process terminates
  - » Acquires termination code from child
  - » Child process is destroyed by kernel
- *Zombie:*– a process that had never been waited for
  - » Hence, cannot go away
  - » See Love, *Linux Kernel Development*, Chapter 3

Processes – Windows

- Windows NT/XP – combines `fork & exec`
  - `CreateProcess(10 arguments)`
  - Not a parent child relationship
  - *Note* – privileges required to create a new process

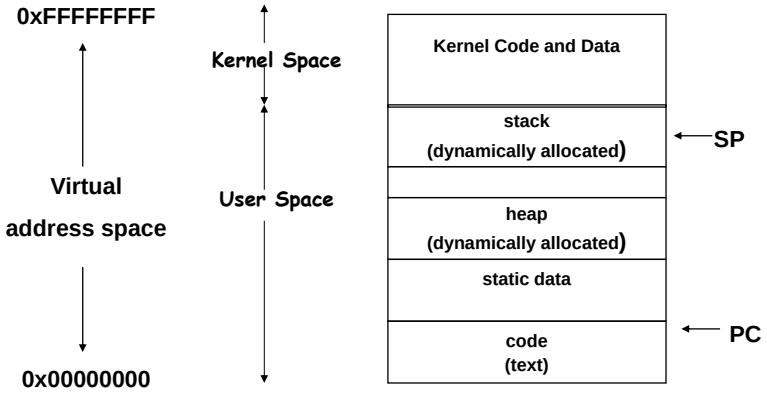
Windows, Unix, and Linux (traditional)

- *Processes* are in *separate* address spaces
  - » By default, no shared memory
- *Processes* are unit of scheduling
  - » A process is *ready*, *waiting*, or *running*
- *Processes* are unit of resource allocation
  - » Files, I/O, memory, privileges, …
- *Processes* are used for (almost) everything!

A Note on Implementation

- Many OS implementations include (parts of) *kernel* in every address space
  - Protected
  - Easy to access
  - Allows *kernel* to see into client processes
    - » Transferring data
    - » Examining state
    - » …

Processes – Address Space



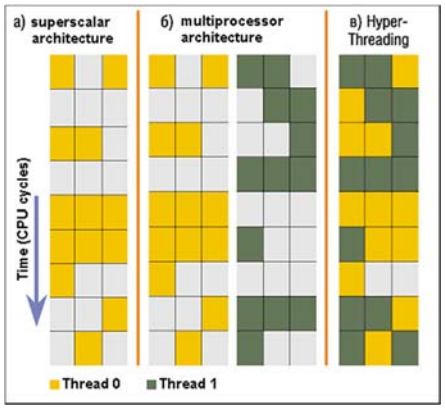
32-bit Linux & Win XP – 3G/1G user space/kernel space

Processes in the OS – Representation

- To users (and other processes) a process is identified by its Process ID (PID)
- In the OS, processes are represented by entries in a Process Table (PT)
  - PID is index to (or pointer to) a PT entry
  - PT entry = Process Control Block (PCB)
- PCB is a large data structure that contains or points to all info about the process
  - Linux - defined in task_struct – over 70 fields
    - » see include/linux/sched.h
  - Windows XP – defined in EPROCESS – about 60 fields

Recall: Modern Technique: SMT/Hyperthreading

- Hardware technique
  - Exploit natural properties of superscalar processors to provide illusion of multiple processors
  - Higher utilization of processor resources
- Can schedule each thread as if were separate CPU
  - However, not linear speedup!
  - If have multiprocessor, should schedule each processor first



- Original technique called "Simultaneous Multithreading"
  - See http://www.cs.washington.edu/research/smt/
  - Alpha, SPARC, Pentium 4 ("Hyperthreading"), Power 5

Thread Level Parallelism (TLP)

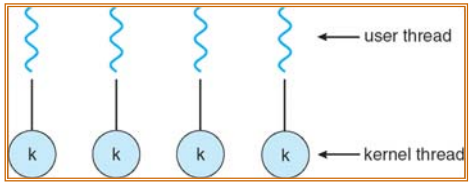
- In modern processors, Instruction Level Parallelism (ILP) exploits implicit parallel operations within a loop or straight-line code segment
- Thread Level Parallelism (TLP) explicitly represented by the use of multiple threads of execution that are inherently parallel
  - Threads can be on a single processor
  - Or, on multiple processors
- Concurrency vs Parallelism
  - Concurrency is when two tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant.
    - » For instance, multitasking on a single-threaded machine.
  - Parallelism is when tasks literally run at the same time, eg. on a multicore processor.
- Goal: Use multiple instruction streams to improve
  - Throughput of computers that run many programs
  - Execution time of multi-threaded programs

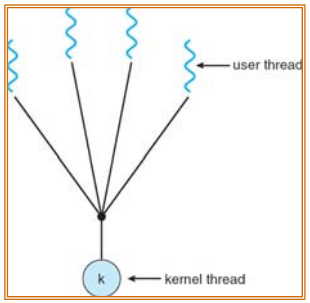
Kernel versus User-Mode threads

- We have been talking about Kernel threads (/tasks)
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- Downside of kernel threads for many OSes: a bit expensive
  - Need to make a crossing into kernel mode to schedule
  - Linux does make *creation* of threads pretty inexpensive
- Even lighter weight option: User Threads
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-premptively relative to each other (only switch on yield())
  - Cheap
- Downside of user threads:
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads
  - Option: *Scheduler Activations*
    - » Have kernel inform user level when thread blocks…

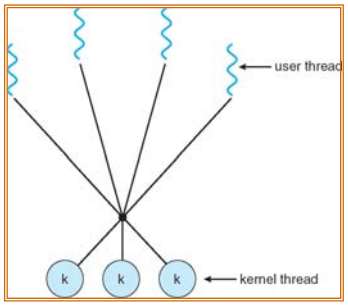
Threading models mentioned by book

Simple One-to-One Threading Model





Many-to-One



Many-to-Many

Common Notions of Thread Creation

- **cobegin/coend**

```
cobegin
    job1(a1);
    job2(a2);
coend
```

  - Statements in block may run in parallel
  - cobegins may be nested
  - Scoped, so you cannot have a missing coend

- **fork/join**

```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```

  - Forked procedure runs in parallel
  - Wait at join point if it's not finished

- **future**

```
v = future(job1(a1));
… = …v…;
```

  - Future expression possibly evaluated in parallel
  - Attempt to use return value will wait

- Threads expressed in the code may not turn into independent computations
  - Only create threads if processors idle
  - Example: Thread-stealing runtimes such as cilk

Overview of POSIX Threads

- Pthreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - Should be relatively uniform across UNIX-like OS platforms
  - Originally IEEE POSIX 1003.1c
- Pthreads contain support for
  - Creating parallelism
  - Synchronizing
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread
    - » Only for HEAP! Stacks not shared

Forking POSIX Threads

Signature:

```
int pthread_create(pthread_t *,
                   const pthread_attr_t *,
                   void * (*)(void *),
                   void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute
                         &thread_fun; &fun_arg);
```

- thread_id is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - Standard default values obtained by passing a NULL pointer
  - Sample attribute: minimum stack size
- thread_fun the function to be run (takes and returns void*)
- fun_arg an argument can be passed to thread_fun when it starts
- errorcode will be set nonzero if the create operation fails

Simple Threading Example (pThreads)

E.g., compile using gcc -lpthread

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Shared Data and Threads

- Variables declared outside of main are shared
- Objects allocated on the heap may be shared (if pointer is passed)
- Variables on the stack are private: passing pointer to these around to other threads can cause problems
- Often done by creating a large "thread data" struct, which is passed into all threads as argument

```
char *message = "Hello World!\n";

pthread_create(&thread1, NULL,
               print_fun,(void*) message);
```


Loop Level Parallelism

- Many application have parallelism in loops

```
double stuff [n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    … pthread_create (…, update_stuff, …,
                      &stuff[i][j]);
```

- But overhead of thread creation is nontrivial
  - update_stuff should have a significant amount of work
- Common Performance Pitfall: Too many threads
  - The cost of creating a thread is 10s of thousands of cycles on modern architectures
  - Solution: Thread blocking: use a small # of threads, often equal to the number of cores/processors or hardware threads

Some More Pthread Functions

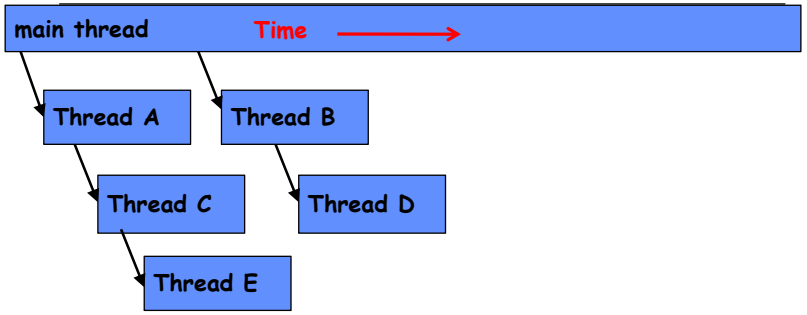
- `pthread_yield();`
  - Informs the scheduler that the thread is willing to yield its quantum, requires no arguments.
- `pthread_exit(void *value);`
  - Exit thread and pass value to joining thread (if exists)
- `pthread_join(pthread_t *thread, void **result);`
  - Wait for specified thread to finish. Place exit value into *result.

Others:

- `pthread_t me; me = pthread_self();`
  - Allows a pthread to obtain its own identifier pthread_t thread;
- `pthread_detach(thread);`
  - Informs the library that the threads exit status will not be needed by subsequent pthread_join calls resulting in better threads performance.

For more information consult the library or the man pages, e.g., man -k pthread..

Thread Scheduling



- Once created, when will a given thread run?
  - It is up to the Operating System or hardware, but it will run eventually, even if you have more threads than cores
  - But – scheduling may be non-ideal for your application
- Programmer can provide hints or affinity in some cases
  - E.g., create exactly P threads and assign to P cores
- Can provide user-level scheduling for some systems
  - Application-specific tuning based on programming model
  - Work in the ParLAB on making user-level scheduling easy to do (Lithe, PULSE)

Multithreaded Execution

- Multitasking operating system:
  - Gives "illusion" that multiple things happening at same time
  - Switches at a course-grained time quanta (for instance: 10ms)
- Hardware Multithreading: multiple threads share processor simultaneously (with little OS help)
  - Hardware does switching
    - » HW for fast thread switch in small number of cycles
    - » much faster than OS switch which is 100s to 1000s of clocks
  - Processor duplicates independent state of each thread
    - » e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - Memory shared through the virtual memory mechanisms, which already support multiple processes
- When to switch between threads?
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

Summary

- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (`yield()`, I/O operations) or involuntary (timer, other interrupts)
- Protection accomplished restricting access:
  - Memory mapping isolates processes from each other
  - Dual-mode for isolating I/O, other resources
- System calls for creating parallelism:
  - fork(), clone()
  - Varying semantics about relationship to parent and parent's address space
- POSIX Threads
  - Higher-level threading interface for programmability across different operating systems