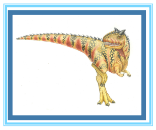
Chapter 4: Threads



Chapter 4: Threads

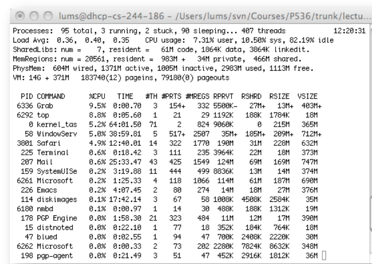
- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples
- Windows XP Threads
- Linux Threads

Objectives

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries
- To examine issues related to multithreaded programming

Processes and Threads

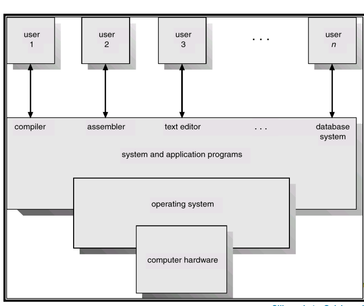
- What is a process?
- What is a thread?



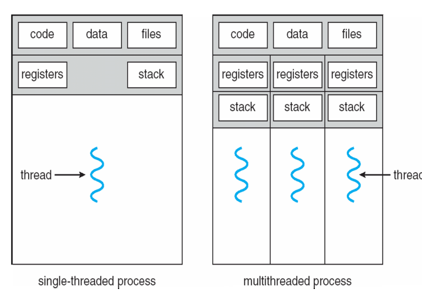
```
lums@dhcp-cs-244-186 - /Users/lums/svn/Courses/P536/trunk/lectu...
Processes:  95 total, 3 running, 2 stuck, 90 sleeping... 407 threads         12:20:31
Load Avg:  0.36,  0.40,  0.35    CPU usage:  7.31% user, 10.50% sys, 82.19% idle
SharedLibs: num =    7, resident =   61M code, 1864K data, 3864K linkedit.
MemRegions: num = 20561, resident =  983M +   34M private,  466M shared.
PhysMem:  604M wired, 1371M active, 1005M inactive, 2983M used, 1113M free.
VM: 14G + 371M   183740(12) pageins, 79180(0) pageouts

  PID COMMAND      %CPU   TIME    #TH #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
 6336 Grab         9.5%  0:00.70   3   154+   332 5500K-   27M+   13M+  403M+
 6292 top          8.8%  0:05.60   1    21     29 1192K   188K  1784K    18M
    0 kernel_tas   5.2% 64:01.50  71     2    824 9060K      0   215M   365M
   58 WindowServ   5.0% 38:59.81   5   517+  2507   35M+  185M+  209M+  712M+
 3801 Safari       4.9% 12:40.01  14   322   1770  190M    31M   228M   632M
  225 Terminal     0.6%  0:18.42   3   111    235 3964K    22M    18M   373M
  207 Mail         0.6% 25:33.47  43   425   1549  124M    69M   169M   747M
  159 SystemUISe   0.2%  3:19.88  11   444    499 8836K    13M    14M   374M
 6261 Microsoft    0.2%  1:25.33   4   118   1066  114M    61M   187M   690M
  226 Emacs        0.2%  4:07.45   2    80    274   14M    18M    27M   376M
  114 diskimages   0.1% 17:42.14   3    67     58 1008K  4508K  2584K    35M
 6180 nmbd         0.1%  0:00.97   1    14     30  488K   188K  1312K    19M
  178 PGP Engine   0.0%  1:58.30  21   323    484   11M    12M    17M   390M
   15 distnoted    0.0%  0:22.10   1    77     18  352K   184K   764K    18M
   47 blued        0.0%  0:02.55   1    94     47  700K  2408K  2220K    30M
 6262 Microsoft    0.0%  0:00.33   2    73    202 2280K  7824K  8632K   348M
  198 pgp-agent    0.0%  0:21.49   3    51     47  452K  2916K  1812K    36M
```


Processes

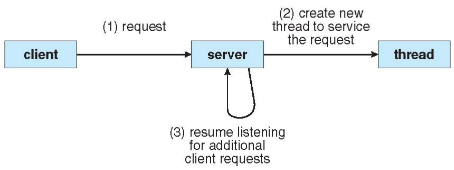
- The process abstraction gave us a way to share computer resources among tasks



Single and Multithreaded Processes

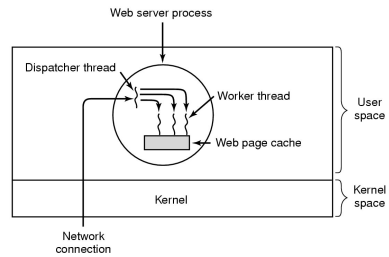


Multithreaded Server Architecture



Example: Web Server

- All incoming requests come to port 80
- Don't want to wait until one request is completely done before handling the next



Web Server: Without threads

- Solution zero
  - Block on incoming requests
  - Service one at a time

```
While(true) {
    Wait_for_work(&buf);
    Do_work(&buf);
}
```


Web Server: Without threads

- Solution zero point five
- Poll (don't block) on incoming requests
- Add incoming requests to pool of work
- Process the work in the pool in some way

```
While(true) {
    Check_for_work(&buf);
    If (work_available(&buf))
        add_to_pool(&buf);
    Do_some_work_in_pool_and_return(&buf);
}
```

- Finite state machine

Web Server: Manager/Worker

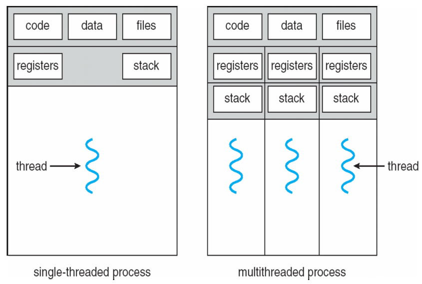
```
while(true) {                       // Manager
    get_next_request(&buf);         // Blocking maybe okay here
    dispatch_to_worker(&buf);
}

while(true) {                       // Worker
    wait_for_work(&buf);
    do_work(&buf);            // Serve pages, e.g.
}
```


Threads and Processes

- What items are associated with a process?
- What are associated with a thread?

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Single and Multithreaded Processes



Benefits

- Responsiveness
- Resource Sharing
- Economy
- Scalability

Multicore Programming

- Multicore systems putting pressure on programmers, challenges include
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

Concurrent Execution on a Single-core System

single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	…

time →

Parallel Execution on a Multicore System

core 1	T_1	T_3	T_1	T_3	T_1	…
core 2	T_2	T_4	T_2	T_4	T_2	…

time →

User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

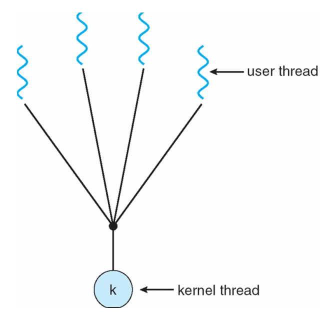
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

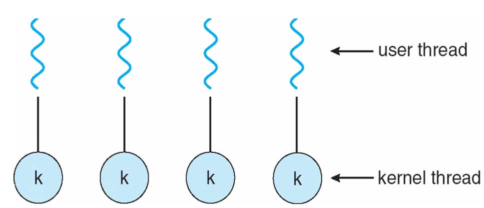
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

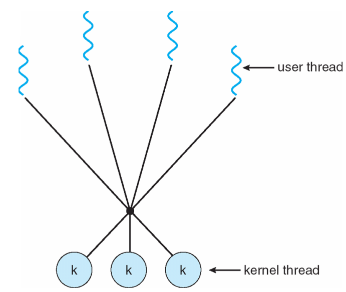
One-to-one Model



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

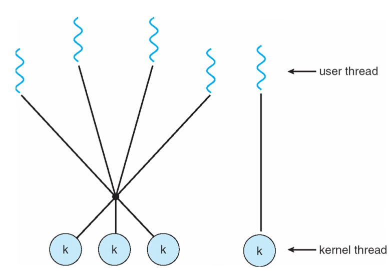
Many-to-Many Model



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

Two-level Model



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads

- Pthreads are created using pthread_create().

```
#include <pthread.h>
int pthread_create (pthread_t *thread_id, const pthread_attr_t *attr
                    void *(*thread_function)(void *), void *argume
```

- Pthreads terminate when the function returns, or the thread calls pthread_exit()

  int pthread_exit (void *status);

- One Thread can wait on the termination of another by using pthread_join()

  int pthread_join (pthread_t thread, void **status_ptr);

- Pthreads also includes many synchronization functions

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

int sum;                          /* this data is shared by the thread(s) */

void *runner(void *param);   /* to be run by the thread */

int main(int argc, char *argv[])
{
  pthread_t tid;                  /* the thread identifier */
  pthread_attr_t attr;            /* set of attributes for the thread */

  pthread_attr_init(&attr);     /* get the default attributes */

  pthread_create(&tid,&attr,runner,argv[1]);    /* create the thread */

  pthread_join(tid,NULL);      /* wait for the thread to exit */

  (void) printf("sum = %d\n",sum);

  return 0;
}
```


Pthreads Example

```
void *runner(void *param)
{
  int i;
  sum = 0;
  int upper = atoi((char *)param);

  if (upper > 0) {
    for (i = 1; i <= upper; i++)
      sum += i;
  }

  pthread_exit(0);
}
```


Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific threa to receive all signals for the process

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

Thread Specific Data

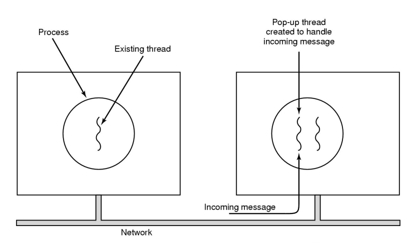
- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Scheduler Activations

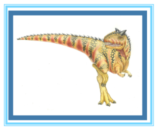
- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

Pop-Up Threads

- Creation of a new thread when message arrives
- New thread is created, not restored (faster)



Aside: Parallel Programming



Fundamental (Program) Design Issues

- Naming: How are logically shared data and/or processes referenced?
- Operations: What operations are provided on these data?
- Ordering: How are accesses to data ordered and coordinated?

Performance (Execution) Issues

- ***Replication***: How are data replicated to reduce communication?
- ***Communication Cost***: Latency, bandwidth, overhead, occupancy

Sequential Programming: Interface

- Naming: Can name any variable (in virtual address space)
  - Hardware (and perhaps compilers) does translation to physical addresses
- Operations: Loads, Stores, Arithmetic, Control
- Ordering: Sequential program order

Sequential Programming: Performance

- Compilers and hardware violate program order without getting caught
  - Compiler: reordering and register allocation
  - Hardware: out of order, pipeline bypassing, write buffers
- Retain dependence order on each "location"
- Transparent replication in caches

Shared Address Programming Model

- Naming: Any thread (process) can name any variable in shared space
- Operations: loads and stores, plus those needed for ordering
- Simplest Ordering Model:
  - Within a process/thread: sequential program order
  - Across threads: some interleaving (as in time-sharing)
  - Additional ordering through explicit synchronization

  - Can compilers/hardware weaken order without getting caught?
    - Different, more subtle ordering models also possible (discussed later)

Synchronization

- Mutual exclusion (locks)
  - Ensure certain operations on certain data can be performed by only one process at a time
  - Room that only one person can enter at a time
  - No ordering guarantees
- Event synchronization
  - Ordering of events to preserve dependences
    - e.g. producer —> consumer of data
  - 3 main types:
    - point-to-point
    - global
    - group

Message Passing Programming Model

- Naming: Processes can name private data directly.
  - No shared address space
- Operations: Explicit communication through send and receive
  - Send transfers data from private address space to another process
  - Receive copies data from process to private address space
  - Must be able to name processes
- Ordering:
  - Program order within a process
  - Send and receive can provide pt to pt synch between processes
  - Mutual exclusion inherent + conventional optimizations legal
- Can construct global address space:
  - Process number + address within process address space
  - But no direct operations on these names

Naming and Operations

- Naming and operations in programming model can be directly supported by lower levels, or translated by compiler, libraries or OS
- Example: Shared virtual address space in programming model
  - Hardware interface supports shared physical address space
    - Direct support by hardware through v-to-p mappings, no software layers
- Hardware supports independent physical address spaces
  - Can provide SAS through OS, so in system/user interface
    - v-to-p mappings only for data that are local
    - remote data accesses incur page faults; brought in via page fault handlers
  - Compilers or runtime, so above sys/user interface

Naming and Operations: Msg Passing

- Direct support at hardware interface
  - But match and buffering benefit from more flexibility
- Support at sys/user interface or above in software
  - Hardware interface provides basic data transport (well suited)
  - Send/receive built in sw for flexibility (protection, buffering)
  - Choices at user/system interface:
    - OS each time: expensive
    - OS sets up once/infrequently, then little sw involvement each time
  - Or lower interfaces provide SAS, and send/receive built on top with buffers and loads/stores
- Need to examine the issues and tradeoffs at every layer
  - Frequencies and types of operations, costs

Ordering

- Message passing: no assumptions on orders across processes except those imposed by send/receive pairs
- Shared address: How processes see the order of other processes' references defines semantics of shared addressing
  - Ordering very important and subtle
  - Uniprocessors play tricks with ordering to gain parallelism or locality
  - These are more important in multiprocessors
  - Need to understand which old tricks are valid, and learn new ones
  - How programs behave, what they rely on, and hardware implications

Replication

- Reduces data transfer/communication
  - depends on naming model
- Uniprocessor: caches do it automatically
  - Reduce communication with memory
- Message Passing naming model at an interface
  - receive replicates, giving a new name
  - Replication is explicit in software above that interface
- SAS naming model at an interface
  - A load brings in data, and can replicate transparently in cache
  - OS can do it at page level in shared virtual address space
  - No explicit renaming, many copies for same name: coherence problem
  - in uniprocessors, "coherence" of copies is natural in memory hierarchy

Communication Performance

- Performance characteristics determine usage of operations at a layer
  - Programmer, compilers etc make choices based on this
- Fundamentally, three characteristics:
  - Latency: time taken for an operation
  - Bandwidth: rate of performing operations
  - Cost: impact on execution time of program
- If processor does one thing at a time: bandwidth μ 1/latency
  - But actually more complex in modern systems
- Characteristics apply to overall operations, as well as individual components of a system

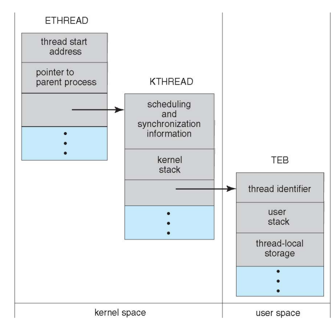
Summary of Design Issues

- Functional and performance issues apply at all layers
- Functional: Naming, operations and ordering
- Performance: Organization
  - latency, bandwidth, overhead, occupancy
- Replication and communication are deeply related
  - Management depends on naming model
- Goal of architects: design against frequency and type of operations that occur at communication abstraction, constrained by tradeoffs from above or below
  - Hardware/software tradeoffs

Operating System Examples

- Windows XP Threads
- Linux Thread

Windows XP Threads



Linux Threads

flag	meaning
`CLONE_FS`	File-system information is shared.
`CLONE_VM`	The same memory space is shared.
`CLONE_SIGHAND`	Signal handlers are shared.
`CLONE_FILES`	The set of open files is shared.

Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

End of Chapter 4