4: Threads

Last Modified:
6/1/2004 11:52:45 AM

Processes

- Recall: A process includes
  - Address space (Code, Data, Heap, Stack)
  - Register values (including the PC)
  - Resources allocated to the process
    - Memory, open files, network connections
- Recall: how processes are created
  - Initializing the PCB and the address space (page tables) takes a significant amount of time
  - Experiment: Time N iterations of fork or vfork
- Recall: Type of interprocess communication
  - IPC is costly also
  - Communication must go through OS ("OS has to guard any doors in the walls it builds around processes for their protection")

Problem needs > 1 independent sequential process?

- Some problems are hard to solve as a single sequential process; easier to express the solution as a collection of cooperating processes
  - Hard to write code to manage many different tasks all at once
  - How would you write code for "make phone calls while making dinner while doing dishes while looking through the mail"
  - Can't be independent processes because share data (your brain) and share resources (the kitchen and the phone)
  - Can't do them sequentially because need to make progress on all tasks at once
  - Easier to write "algorithm" for each and when there is a lull in one activity let the OS switch between them
  - Let OS manage the waiting and multitasking
- On a multiprocessor, exploit parallelism in problem

Example: Web Server

- Web servers listen on an incoming socket for requests
  - Once it receives a request, it ignore listening to the incoming socket while it services the request
  - Must do both at once
- One solution: Create a child process to handle the request and allow the parent to return to listening for incoming requests
- Problem: This is inefficient because of the address space creation (and memory usage) and PCB initialization

Observation

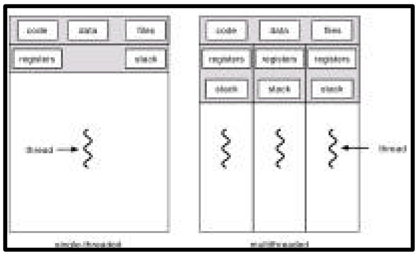
- There are similarities in the process that are spawned off to handle requests
  - They share the same code, have the same privileges, share the same resources (html files to return, cgi script to run, database to search, etc.)
- But there are differences
  - Operating on different requests
  - Each one will be in a different stage of the "handle request" algorithm

Idea

- Let these tasks share the address space, privileges and resources
- Give each their own registers (like the PC), their own stack etc

- Process – unit of resource allocation (address space, privileges, resources)
- Thread – unit of execution (PC, stack, local variables)

Single-Threaded vs Multithreaded Processes



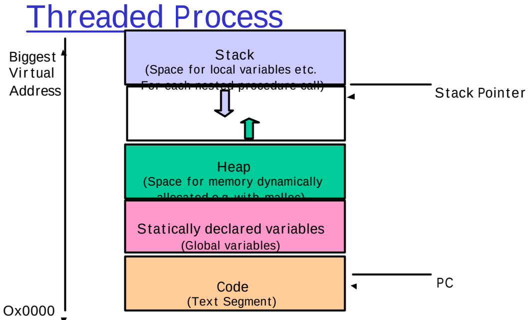
-7

Process vs Thread

- Each thread belongs to one process
- One process may contain multiple threads
- Threads are logical unit of scheduling
- Processes are the logical unit of resource allocation

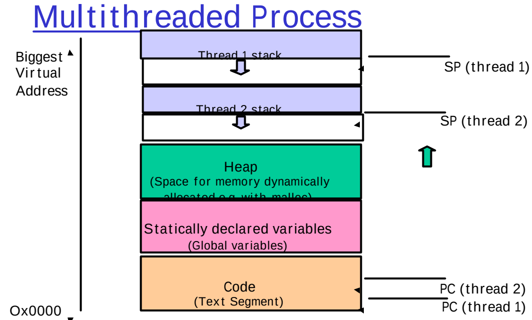
-8

Address Space Map For Single-Threaded Process



-9

Address Space Map For Multithreaded Process



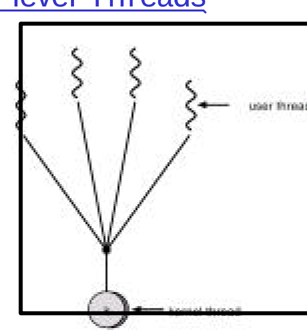
-10

Kernel support for threads?

- Some OSes support the notion of multiple threads per process and others do not
- Even if no "kernel threads" can build threads at user level
  - Each "multi-threaded" program gets a single kernel in the process
  - During its timeslice, it runs code from its various threads
  - User-level thread package schedules threads on the kernel level process much like OS schedules processes on the CPU
    - SAT question? CPU is to OS is to processes like?
    - Kernel thread is to User-level thread package is to user threads
  - User-level thread switch must be programmed in assembly (restore of values to registers, etc.)

-11

User-level Threads



-12

User-level threads

- How do user level thread packages avoid having one thread monopolize the processes time slice?
  - Solve much like OS does
- Solution 1: Non-preemptive
  - Rely on each thread to periodically yield
  - Yield would call the scheduling function of the library
- Solution 2: OS is to user level thread package like hardware is to OS
  - Ask OS to deliver a periodic timer signal
  - Use that to gain control and switch the running thread

-13

Kernel vs User Threads

- One might think, kernel level threads are best and only if kernel does not support threads use user level threads
- In fact, user level threads can be much faster
  - Thread creation , "Context switch" between threads, communication between threads all done at user level
  - Procedure calls instead of system calls (verification of all user arguments, etc.) in all these cases!

-14

Problems with User-level threads

- OS does not have information about thread activity and can make bad scheduling decisions
- Examples:
  - If thread blocks, whole process blocks
    - Kernel threads can take overlap I/O and computation within a process!
  - Kernel may schedule a process with all idle threads

-15

Scheduler Activations

- If have kernel level thread support available then use kernel threads *and* user-level threads
- Each process requests a number of kernel threads to use for running user-level threads on
- Kernel promises to tell user-level before it blocks a kernel thread so user-level thread package can choose what to do with the remaining kernel level threads
- User level promises to tell kernel when it no longer needs a given kernel level thread

-16

Thread Support

- Pthreads is a user-level thread library
  - Can use multiple kernel threads to implement it on platforms that have kernel threads
- Java threads (extend Thread class) run by the Java Virtual Machine
- Kernel threads
  - Linux has kernel threads (each has its own task_struct) – created with clone system call
  - Each user level thread maps to a single kernel thread (Windows 95/98/NT/2000/XP, OS/2)
  - Many user level threads can map onto many kernel level threads like scheduler activations (Windows NT/2000 with ThreadFiber package, Solaris 2)

-17

Pthreads Interface

- POSIX threads, user-level library supported on most UNIX platforms
- Much like the similarly named process functions
  - thread = pthread_create(procedure)
  - pthread_exit
  - pthread_wait(thread)

Note: To use pthreads library,
 #include <pthread.h>
 compile with -lpthread

-18

Pthreads Interface (con't)

- Pthreads support a variety of functions for thread synchronization/coordination
  - Used for coordination of threads (ITC ☺) – more on this soon!
- Examples:
  - Condition Variables (pthread_cond_wait, pthread_signal)
  - Mutexes(pthread_mutex_lock, pthread_mutex_unlock)

-19

Performance Comparison

Processes	Fork/Exit	251
Kernel Threads	Pthread_create/ Pthread_join	94
User-level Threads	Pthread_create/ Pthread_join	4.5

In microseconds, on a 700 MHz Pentium, Linux 2.2.16, Steve Gribble, 2001.

-20

Windows Threads

HANDLE CreateThread(
LPSECURITY_ATTRIBUTES *lpThreadAttributes*,
DWORD *dwStackSize*,
LPTHREAD_START_ROUTINE *lpStartAddress*,
DWORD *dwCreationFlags*,
LPVOID *lpParameter*,
DWORD *dwCreationFlags*,
LPDWORD *lpThreadId***);**

-21

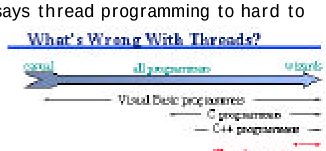
Windows Thread Synchronization

- Windows supports a variety of objects that can be used for thread synchronization
- Examples
  - Events (CreateEvent, SetEvent, ResetEvent, WaitForSingleObject)
  - Semaphores (CreateSemaphore, ReleaseSemaphore, WaitForSingleObject)
  - Mutexes (CreateMutex, ReleaseMutex, WaitForSingleObject)
  - WaitForMultipleObject
  - More on this when we talk about synchronization

-22

Warning: Threads may be hazardous to your health

- One can argue (and John Ousterhout did) that threads are a bad idea for most purposes
- Anything you can do with threads you can do with an event loop
  - Remember "make phone calls while making dinner while doing dishes while looking through the mail"
- Ousterhout says thread programming to hard to get right



-23

Outtakes

- Processes that just share code but do not communicate
  - Wasteful to duplicate
  - Other ways around this than threads

-24

Example: User Interface

- Allow one thread to respond to user input while another thread handles a long operation
- Assign one thread to print your document, while allowing you to continue editing

-25

Benefits of Concurrency

- Hide latency of blocking I/O without additional complexity
  - Without concurrency
    - Block whole process
    - Manage complexity of asynchronous I/O (periodically checking to see if it is done so can finish processing)
- Ability to use multiple processors to accomplish the task
- Servers often use concurrency to work on multiple requests in parallel
- User Interfaces often designed to allow interface to be responsive to user input while servicing long operations

-26

Thread pools

- What they are and how they avoid thread creation overhead

-27

Experiment

- Start up various processes under Windows (Word, IE,..)
- How many processes are started?
- How many threads and of what priority?

-28