Threads

Announcements

Cooperating Processes

- Processes can be independent or work cooperatively
- Cooperating processes can be used:
  - to gain speedup by overlapping activities or working in parallel
  - to better structure an application as set of cooperating processes
  - to share information between jobs
- Sometimes processes are structured as a pipeline
  - each produces work for the next stage that consumes it

Case for Parallelism

```
main()
 read_data()
 for(all data)
   compute();
   write_data();
 endfor
```

```
main()
   read_data()
   for(all data)
        compute();
        CreateProcess(write_data());
   endfor
```

Case for Parallelism

Consider the following code fragment

```
for(k = 0; k < n; k++)
   a[k] = b[k] * c[k] + d[k] * e[k];
```

```
CreateProcess(fn, 0, n/2);
CreateProcess(fn, n/2, n);
fn(l, m)
   for(k = l; k < m; k++)
        a[k] = b[k] * c[k] + d[k] * e[k];
```

Case for Parallelism

Consider a Web server

create a number of processes, and for each process do:

- get network message from client
- get URL data from disk
- compose response
- send response

Processes and Threads

- A full process includes numerous things:
  - an address space (defining all the code and data pages)
  - OS resources and accounting information
  - a "thread of control",
    - defines where the process is currently executing
    - That is the PC and registers
- Creating a new process is costly
  - all of the structures (e.g., page tables) that must be allocated
- Communicating between processes is costly
  - most communication goes through the OS

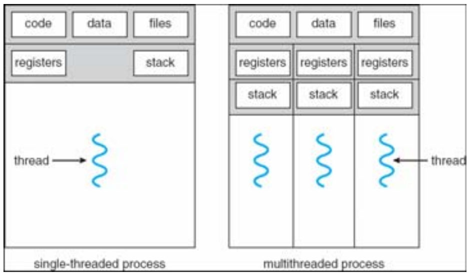
"Lightweight" Processes

- Idea: why don't we separate the idea of process (address space, accounting, etc.) from that of the minimal "thread of control" (PC, SP, registers)?
- Like our "heavyweight" processes:
  - Each has its own PC, registers, and stack pointer
- Unlike our "heavyweight" processes:
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They share almost everything in the process

Threads and Processes

- Most operating systems therefore support two entities:
  - the process,
    - which defines the address space and general process attributes
  - the thread,
    - which defines a sequential execution stream within a process
- A thread is bound to a single process.
  - For each process, however, there may be many threads.
- Threads are the unit of scheduling
- Processes are *containers* in which threads execute

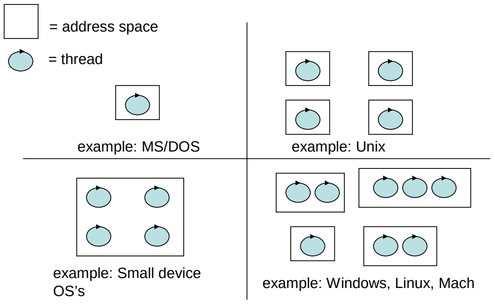
Multithreaded Processes



Threads vs. Processes

A thread has no data segment or heap	A process has code/data/heap & other segments
A thread cannot live on its own, it must live within a process	There must be at least one thread in a process
There can be more than one thread in a process, the first thread calls main & has the process's stack	*Threads within a process share code/data/heap, share I/O, but each has its own stack & registers*
Inexpensive creation	Expensive creation
Inexpensive context switching	Expensive context switching
If a thread dies, its stack is reclaimed	If a process dies, its resources are reclaimed & all threads die

How OSes support threads?



= address space

= thread

example: MS/DOS

example: Unix

example: Small device OS's

example: Windows, Linux, Mach

Cooperative Threads

Each thread runs until *it* decides to give up the CPU

```
main()
{
    tid t1 = CreateThread(fn, arg);
    …
    Yield(t1);
}
fn(int arg)
{
    …
    Yield(any);
}
```

Cooperative Threads

- Cooperative threads use non pre-emptive scheduling

- Advantages:
  - Simple
    - Small, real-time OSs
- Disadvantages:
  - For badly written code
- Scheduler gets invoked only when Yield is called
- A thread could yield the processor when it blocks for I/O

Non-Cooperative Threads

- No explicit control passing among threads
- Rely on a scheduler to decide which thread to run
- A thread can be pre-empted at any point
- Often called pre-emptive threads
- Most modern thread packages use this approach

Kernel Threads

- Also called Lightweight Processes (LWP)
- Kernel threads still suffer from performance problems
- Operations on kernel threads are slow because:
  - a thread operation still requires a system call
  - kernel threads may be overly general
    - to support needs of different users, languages, etc.
  - the kernel doesn't trust the user
    - there must be lots of checking on kernel calls

User-Level Threads

- For speed, implement threads at the user level
- A user-level thread is managed by the run-time system
  - user-level code that is linked with your program
- Each thread is represented simply by:
  - PC
  - Registers
  - Stack
  - Small control block
- All thread operations are at the user-level:
  - Creating a new thread
  - switching between threads
  - synchronizing between threads

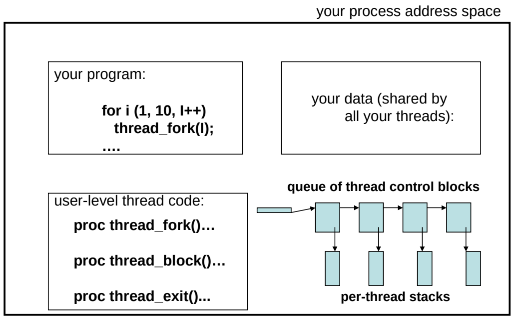
User-Level Threads

- User-level threads
  - the thread scheduler is part of a library, outside the kernel
  - thread context switching and scheduling is done by the library
  - Can either use cooperative or pre-emptive threads
    - cooperative threads are implemented by:
      - CreateThread(), DestroyThread(), Yield(), Suspend(), etc.
    - pre-emptive threads are implemented with a timer (signal)
      - where the timer handler decides which thread to run next

Example User Thread Interface

t = thread_fork(initial context)
    create a new thread of control
thread_stop()
    stop the calling thread, sometimes called thread_block
thread_start(t)
    start the named thread
thread_yield()
    voluntarily give up the processor
thread_exit()
    terminate the calling thread, sometimes called thread_destroy

Key Data Structures



Multiplexing User-Level Threads

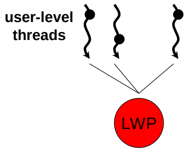
- The user-level thread package sees a "virtual" processor(s)
  - it schedules user-level threads on these virtual processors
  - each "virtual" processor is implemented by a kernel thread
- The big picture:
  - Create as many kernel threads as there are processors
  - Create as many user-level threads as the application needs
  - Multiplex user-level threads on top of the kernel-level threads
- Why not just create as many kernel-level threads as app needs?
  - Context switching
  - Resources

User-Level vs. Kernel Threads

User-Level	Kernel-Level
• Managed by application	• Managed by kernel
• Kernel not aware of thread	• Consumes kernel resources
• Context switching cheap	• Context switching expensive
• Create as many as needed	• Number limited by kernel resources
• Must be used with care	• Simpler to use

Key issue: kernel threads provide virtual processors to user-level threads, but if all of kthreads block, then all user-level threads will block *even* if the program logic allows them to proceed

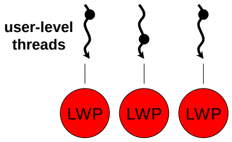
Many-to-One Model



Thread creation, scheduling, synchronization done in user space.
Mainly used in language systems, portable libraries

👍Fast - no system calls required
👍Few system dependencies; portable
👎No parallel execution of threads - can't exploit multiple CPUs
👎All threads block when one uses synchronous I/O

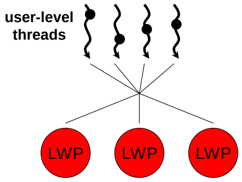
One-to-one Model



Thread creation, scheduling, synchronization require system calls
Used in Linux Threads, Windows

👍 More concurrency
👍 Better multiprocessor performance
👎 Each user thread requires creation of kernel thread
👎 Each thread requires kernel resources; limits number of total threads

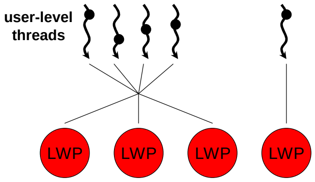
Many-to-Many Model



If U < L? No benefits of multithreading
If U > L, some threads may have to wait for an LWP to run
- Active thread - executing on an LWP
- Runnable thread - waiting for an LWP

A thread gives up control of LWP under the following:
– synchronization, lower priority, yielding, time slicing

Two-level Model



- Combination of one-to-one + “strict” many-to-many models
- Supports both bound and unbound threads
  - Bound threads - permanently mapped to a single, dedicated LWP
  - Unbound threads - may move among LWPs in set
- Thread creation, scheduling, synchronization done in user space
- Flexible approach, “best of both worlds”
- Used in Solaris implementation of Pthreads and several other Unix implementations (IRIX, HP-UX)

Multithreading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
  - Asynchronous vs. Deferred Cancellation
- Signal handling
  - Which thread to deliver it to?
- Thread pools
  - Creating new threads, unlimited number of threads
- Thread specific data
- Scheduler activations
  - Maintaining the correct number of scheduler threads

Thread Hazards

```
int a = 1, b = 2, w = 2;
main() {
    CreateThread(fn, 4);
    CreateThread(fn, 4);
    while(w)  ;
}
fn() {
   int v = a + b;
   w--;
}
```

Concurrency Problems

A statement like w-- in C (or C++) is implemented by several machine instructions:

```
ld      r4, #w
add     r4, r4, -1
st      r4, #w
```

Now, imagine the following sequence, what is the value of w?

```
ld      r4, #w
______________                  ______________
______________                  ld      r4, #w
______________                  add     r4, r4, -1
add     r4, r4, -1              st      r4, #w
st      r4, #w
```