Review

- The Joys and Pains of Threads and Multithreading
  - what is a thread
  - threads vs. processes
  - opportunities and risks

Outline

- More Joys and Pains

Threads & Multithreading

(*Continued*)

Thread Hazards

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

What happens here?
Lesson 7: Concurrency

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
```

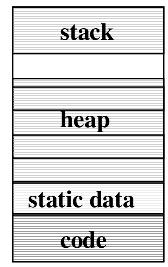
Now, imagine the following sequence, what is the value of w?

ld	r4, #w	C.S.	______________	
______________			ld	r4, #w
______________			add	r4, r4, -1
______________			st	r4, #w
add	r4, r4, -1			
st	r4, #w			

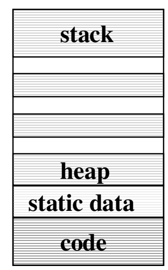
Thread Hazards

- In a classical process (only one thread), the process's stack segment serves to be the stack of the root thread:
  - The stack segment is far away from the code and data segments
  - It grows implicitly during function calls up to a maximum size
- In a multithreaded process, each thread needs a stack
  - Where should the stack be?
  - What size?
  - One important restriction: A thread's stack has to be part of the process's address space

Thread Hazards



- Option 1:
  - allocate the stacks on the heap (using malloc)
  - easier, hazardous



- Option 2:
  - allocate the stacks away from the other segments
  - require OS support, safer

Other Hazards?

- If there is a signal, who should get it?
  - One?
  - All?
  - Some designated thread?
- If there is an exception:
  - Kill only offending thread?
  - Kill process?
- Hidden concurrency problems:
  - Sharing through files

Signaling

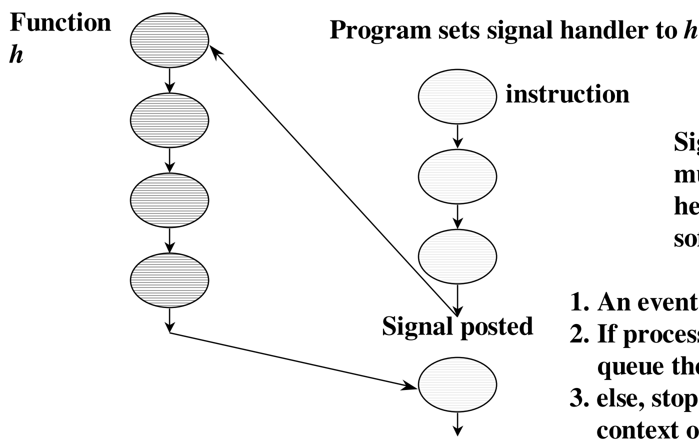
- The O.S. translates some events into *asynchronous* signals:
  - input/output
  - alarms
- The O.S. also translates exceptions into signals, e.g.:
  - Memory access violations, illegal instructions, overflow, etc.
- Signals also used for process control, e.g.:
  - SIGKILL to kill a process, SIGSTOP to stop a process, etc.
- Rudimentary process communications, e.g.:
  - processes may use the *kill*() system call to send signals to each other (SIGUSR1, SIGUSR2)

Signal Handling

- A process can:
  - rely on the default signal handler (e.g. core dump in UNIX)
  - ignore signal (temporarily or permanently, dangerous)
  - sets its own handler
- Fundamental difference between signals & regular I/O
  - A process gets regular I/O when it chooses to do so (synchronous)
  - A process is interrupted and forced to handle a signal whenever a signal is posted (and not ignored). This is asynchronous

Signals are very difficult to deal with and are better avoided

How Does it Work?



Signals work very much like interrupts hence the name software interrupts

1. **An event of interest occurs**
2. **If process masks event out queue the signal**
3. **else, stop process, push its context on stack, force process to jump to handler**

Thread Variations

Threads differ according to three dimensions:

- Implementation: kernel-level versus user-level
- Execution: uniprocessor vs. multi-processor
- cooperative vs. uncooperative

This gives up to 8 combinations

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

- By their nature, cooperative threads use non pre-emptive scheduling (e.g. Windows 3.1)
- Advantages:

- Disadvantages:

- The scheduler gets invoked only when Yield is called with the argument *any* (or when a thread blocks)
- Depending on the thread package semantics, 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

Execution on Uni vs Multi Processors

- Programmers often “exploit” the fact that a program is designed to run on a single processor to simplify the solution to problems such as concurrency control, etc.
- However, this is very bad programming style
- VALUABLE ADVICE: Always write your multithreaded program as if it were to run on a true, honest-to-goodness multiprocessor

Kernel Threads

- Simply stated, a kernel thread is one that is implemented and supported by the kernel (often called lightweight processes (LWP))
- Each thread has its “Thread Control Block” (tcb)
- The tcb becomes the unit of scheduling in the system
- It is similar in “spirit” to the pcb, but contains much less information
- tcb contains:
  - placeholder for the context
  - the thread id
  - queueing support
  - thread state
- What are the modifications that are needed now to the pcb?

User-Level Threads

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

User-Level Threads

- Context switching in user space?
  - Essentially the same as the implementation in the kernel, save registers in the tcb (in user memory), bring the new context from the corresponding tcb, and "jump" to the program counter location of the new thread
- The scheduler can implement any scheduling algorithm as before
- Caveat-Emptor: The kernel knows absolutely NOTHING about user-level threads
  - the user-level threads actually multiplex themselves on the kernel-level threads
  - the kernel only sees the kernel-level threads that it gave to a process

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 actually 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 would you want to do that? Why not just create as many kernel-level threads as the application needs?
  - Context switching
  - Resources

User-Level vs. Kernel Threads

User-Level
- Managed by application
- Kernel is not aware of thread
- Context switching done by application (cheap)
- Can create as many as the application needs
- Must be used with care

Kernel-Level
- Managed by kernel
- Consumes kernel resources
- Context switching done by kernel (expensive)
- Number limited by kernel resources
- 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

Retrospect on Scheduling

- Scheduling threads is very similar to scheduling processes:
  - it could be pre-emptive or non pre-emptive
  - it could use any scheduling algorithm (FCFS, SJF, RR, …)
  - threads (not processes) now get to be on the ready queue, can be blocked, or can be running, etc.
- But:
  - a good scheduler takes into account the relation between threads and processes
  - Therefore, we have several variations

Thread Scheduling

Since all threads share code & data segments

- Option 1: Ignore this fact
- Option 2: Gang scheduling-- run all threads belonging to a process together (multiprocessor only)
  - if a thread needs to synchronize with another thread, the other one is available and active
- Option 3: Two-level scheduling-- schedule processes, and within each process, schedule threads
  - reduce context switching overhead and improve cache hit ratio
- Option 4: Space-based affinity-- assign threads to processors (multiprocessor only)
  - improve cache hit ratio, but can bite under low-load condition