Threads

- A thread is similar to a process
  - sometimes called a lightweight process
  - several threads (of control) can execute within the same address space

- Like a process, a thread
  - is a basic unit of CPU utilization
  - represents the state of a program
  - can be in one of several states: ready, blocked, running, or terminated
  - has its own program counter, registers, and stack
  - executes sequentially, can create other threads, block for a system call

- Unlike a process, a thread
  - shares with peer threads, its code section, data section, and operating-system resources such as open files and signals
  - is simpler and faster

Outline

- Announcements
  - Demos today and tomorrow
    - Send writeups in plain text: no fancy formatting/attachments
  - Questions?

- Threads
  - What are they
  - Multithreading models

- Process Cooperation
  - Why required
  - Shared memory and message passing
  - Critical sections

[Silberschatz/Galvin/Gagne: 5.1-5.5, 7.1-7.2]
Threads: Why Simpler?

Threads share the process address space

- **Benefits for the user:**
  - communication is easier
  - communication is more efficient
  - security may not be necessary
    assumed to operate within the same protection domain
  - one blocking thread need not block other threads in the process

- **Benefits for the OS**
  - context switching is more efficient
    memory mappings can remain unchanged
    cache need not be flushed
  - can run a process across multiple nodes of a multiprocessor
    performance advantages if threads can execute in parallel (e.g., web servers)

Types of Threads

- **User-level threads** (e.g., pthreads: Section 5.4, Java threads: Section 5.8)
  - OS does not know about them
  - implemented/scheduled by library routines
  - blocking operations block the entire process (even with ready threads)
  - operations based on local criteria may be less effective (e.g., scheduling)

- **Kernel-level threads** (e.g., Solaris 2: Section 5.5, Win2k: Section 5.6)
  - known to the OS
  - scheduled by the OS
  - process need not block if one of its threads blocks on a system call
  - thread operations are expensive
    switching threads involves kernel interaction (via an interrupt)
  - the kernel can do a better job of allocating resources

Multithreading Models

- Most systems provide support for both user and kernel threads

Three dominant models for mapping threads to kernel resources

- **Many-to-one**
  - Thread management done in user space
  - Entire process blocks if a thread does a blocking operation
  - E.g., systems without kernel threads

- **One-to-one**
  - Each user-thread mapped to a kernel thread
  - Allows more concurrency
  - E.g., Windows 2000 (fibers: many-to-one)

- **Many-to-many**
  - Combination of the above two
  - E.g., Solaris 2

POSIX Threads (pthreads)

- A portable API for multithreaded programs
  - Some pthreads implementations do map threads to kernel threads
  - Most rely on user-level threading support
    - Assembly instructions to save/restore registers

- Calls for creating, exiting, joining pthreads
  - `pthread_create:` start execution of this thread
    - Takes function pointer as an argument
  - `pthread_exit:` terminate execution of this thread
  - `pthread_join:` wait for a particular thread to exit

- Other calls
  - Help set thread attributes (stack size, scheduling behavior, etc.)
  - Specify signal handling
    - Signals are a way of allowing processes to respond to events
    - Interrupts (Ctrl-C), others
    - Multithreaded systems need to define a way for signals to be communicated to individual threads (see Section 5.3.3)
      - All threads, a specific thread, only those threads that do not block the signal, …
Processes and Threads in Solaris 2

- OS schedules execution of kernel threads (KTs)
  - runs them on the CPUs
  - a KT can be pinned to a CPU
- A task consists of one or more lightweight processes (LWPs)
  - LWPs in a task may
    - contain several user-level threads
    - issue a system call
    - block
- A LWP is associated with a KT
- There are KTs with no LWP

Outline

- Announcements
  - Demos today and tomorrow
  - Send writeups in plain text: no fancy formatting/attachments
  - Questions?
- Threads
  - What are they
  - Multithreading models
- Process Cooperation
  - Why required
  - Shared memory and message passing
  - Critical sections

[Silberschatz/Galvin/Gagne: 5.1-5.5, 7.1-7.2]

Process Cooperation

- Why do processes cooperate?
  - modularity: breaking up a system into several sub-systems
    - e.g.: an interrupt handler and device driver that need to communicate
  - convenience: users might want to have several processes share data
  - speedup: a single program is run as several sub-programs
- How do processes cooperate?
  - communication abstraction: producers and consumers
    - producers produce a piece of information
    - consumers use this information
  - abstraction helps deal with general “phenomena” and simplifies correctness arguments
- Two general classes of process cooperation techniques
  - shared memory
  - message passing

Shared Memory (Procedure-oriented System)

- Processes can directly access data written by other processes
  - examples: POSIX threads, Java, Mesa, small multiprocessors
- A finite-capacity shared buffer

```c
N: integer -- buffer size
nextin = nextout = 1 initially; -- start of buffer
buffer: array of size N

Producer:
  Repeat
  -- produce an item in tempin
  while (nextin+1) mod n = nextout do wait-a-bit;
  buffer[nextin] := tempin;
  nextin := (nextin+1) mod n;

Consumer:
  Repeat
  while nextin = nextout do wait-a-bit;
  tempout := buffer[nextout];
  nextout := (nextout+1) mod n;
  -- consume the item in tempout
```
Message Passing (Message-oriented System)

- Execution is in separate address spaces
  - communication using message channels
  - examples: UNIX processes, large multiprocessors, etc.

- Components
  - messages and message identifiers
  - message channels and ports
    • channels (pipes) must be bound to ports
    • queues associated with ports
  - message transmission operations
    • SendMessage(channel, body) returns id
    • AwaitReply(id)
    • RecvMessage(port) returns id
    • SendReply(id, body)

- Many variants: See Section 4.5
  ➤ Focus on shared memory for next few lectures

Bounded Buffers Using Counters

```plaintext
N: integer -- buffer size
counter: integer = 0 initially;
nextin = nextout = 1 initially; -- start of buffer
buffer: array of size N

Producer:
  Repeat
  -- produce an item in tempin
  while counter = N do wait-a-bit;
  buffer[nextin] := tempin;
  nextin := (nextin+1) mod n;
  counter := counter+1;

Consumer:
  Repeat
  while counter = 0 do wait-a-bit;
  tempout := buffer[nextout];
  nextout := (nextout+1) mod n;
  counter := counter-1;
  -- consume the item in tempout
```

Interleaving of Increment/Decrement

- Each of increment and decrement are actually implemented as a series
  of machine instructions on the underlying processor

```plaintext
Producer
  register1 := counter
  register1 := register1 + 1
  counter := register1

Consumer
  register2 := counter
  register2 := register2 - 1
  counter := register2

An interleaving
  counter = 5; a producer followed by a consumer

Producer
  register1 := counter
  register1 := register1 + 1

Consumer
  {register1 = 5}
  {counter = 6}
  {counter = 4}
```

The Problem

- Increment and decrement are not atomic or uninterruptable
  - two or more operations are executed atomically if the result of their
    execution is equivalent to that of some serial order of execution
  - operations which are always executed atomically are called atomic
    • byte read; byte write;
    • word read; word write

- The code containing these operations creates a race condition
  - produces inconsistencies in shared data

- Reasons for non-atomic execution
  - interrupts
  - context-switches
The Solution

- The producer and consumer processes need to synchronize — so that they do not access shared variables at the same time
  - this is called mutual exclusion
    - the shared and critical variables can be accessed by only one process at a time
    - access must be serialized even if the processes attempt concurrent access
      - in the previous example: counter increment and decrement operations
- General framework for achieving this: Critical Sections
  - work independent of the particular context or need for synchronization

Critical Sections

- Critical sections: General framework for process synchronization
  
  ENTRY-SECTION
  CRITICAL-SECTION-CODE
  EXIT-SECTION

  - the ENTRY-SECTION controls access to make sure that no more than one process $P_i$ gets to access the critical section at any given time
    - acts as a guard
  - the EXIT-SECTION does bookkeeping to make sure that other processes that are waiting know that $P_i$ has exited

- How can we implement critical sections?
  - turn off interrupts around critical operations
  - build on top of atomic memory load/store operations
  - provide higher-level primitives

Two-Process Solutions: Turn Counters

- Shared integer variable: $turn$ (initialized to 0)
  - for $i \in \{0, 1\}$: $P_i$ executes:
    
    while ($turn \neq i$) wait-a-bit;
    CRITICAL SECTION;
    $turn := j$;
  
  - the while loop is the entry section
    - process $P_i$ waits till its turn occurs
  
  - the single instruction $turn := j$ constitutes the exit section
    - informs the other process of its turn

- Mutual exclusion?
  - assume atomic loads and stores

- Drawbacks?
  - if $P_i$ never wants to execute the critical section, $P_i$ cannot reenter;
    - access must alternate

Two-Process Solutions: Array of Flags

- Boolean array $flag$ (initialized to false), $P_i$ executes:
  
  1: $flag[i] := true$;
  2: while $flag[j]$ wait-a-bit;
  3: $flag[i] := false$;

- Mutual exclusion?

- Is this good enough?
  - No: $P_0$ and $P_1$ can be looping on instruction 2 forever
Criteria for Correctness

Three conditions
- **Mutual exclusion**
- **Progress**
  - at least one process requesting entry to a critical section will be able to enter it if there is no other process in it
- **Bounded waiting**
  - no process waits indefinitely to enter the critical section once it has requested entry

Two-Process Solutions: Petersen’s Algorithm

Combines the previous two ideas

```
1: flag[i] := true
2: turn := j
3: while (flag[j] and (turn == j)) wait-a-bit
   CRITICAL SECTION
4: flag[i] := false
```

Does the algorithm satisfy the three criteria?

Petersen’s Algorithm: Mutual Exclusion

```
1: flag[i] := true
2: turn := j
3: while (flag[j] and (turn == j)) wait-a-bit
   CRITICAL SECTION
4: flag[i] := false
```

Suppose: P₀ is in its critical section, and P₁ is wanting to enter
This can happen only if either
- (case 1) P₀ found flag[1] false, or
- (case 2) P₀ found turn == 0
  - in the first case: P₁ will set turn after P₀ did, and find turn == 0
  - in the second case: P₁ has already set turn = 0
  - in both cases: P₁ will wait till flag[0] == false

Petersen’s Algorithm: Progress and Bounded Waiting

```
1: flag[i] := true
2: turn := j
3: while (flag[j] and (turn == j)) wait-a-bit
   CRITICAL SECTION
4: flag[i] := false
```

To prove progress:
- if P₁ is not ready to enter the critical section
  * flag[1] will be false ⇒ P₀ can enter

To prove bounded waiting:
- let P₀ be in the critical section and P₁ be waiting on instruction 3 above
  - if P₀ exits and goes elsewhere,
    * either P₁ will find flag[0] to be false
    * if not, P₀ will attempt to reenter the critical section, setting turn := j
  - in either case, P₁ will find the condition for waiting in (3) to be false and will enter the critical section
Can These Solutions be Extended to >2 Processes?

- N-process solutions
  - do exist: Bakery Algorithm (see Section 7.2.2)
  - but reasoning gets even more complicated!

- So, we can implement critical sections using only support for atomic memory loads and stores
- But, there **must** be an easier way!

- Higher-level synchronization primitives
  - locks (mutexes), semaphores, condition variables
  - rely on more support from hardware
    - disabling of interrupts: only around the primitives
    - atomic read-modify-write operations