(Review) Process Scheduling: Three Concerns

- What happens to the process currently using the CPU?
  - Process context switch
- How do you keep track of what each process should be doing?
  - Process queues
- How do you decide which process does what?
  - Schedulers

Look at the Nachos code: Thread::Yield, SWITCH
Nachos Lab 1: Consequences of asynchronous context switches
Concern 2: Process Queues

- **state transition**
  - internal
  - external
- **PCBs**
- **new**
- **create**
- **time-out**
- **suspend**
- **kill**
- **exit**
- **event or resource available**
- **PCBs**
- **stopped**
- **dispatch**
- **running**
- **Ready**
- **running**
- **dispatch**
- **terminated**
- **kill**
- **event or resource wall**
- **error**
- **PCBs**

Concern 3: Schedulers

- **The long-term scheduler**
  - **operation**: creates processes and adds them to the ready queue
  - **frequency**: infrequent, ~minutes
  - **objective**: maintain good throughput by ensuring mix of I/O and CPU jobs

- **The short-term scheduler**
  - **operation**: allocates CPU and other resources to ready jobs
  - **frequency**: frequent, ~100 ms (a context switch takes ~10s of μsecs)
  - **objective**: ensure good response times in time-sharing systems

- **The medium-term scheduler**
  - **operation**: swaps some processes out of the short-term scheduler’s loop
  - **frequency**: somewhere between the short- and long-term schedulers
  - **objective**: to prevent over-multiprogramming (thrashing)
    - required when the long-term scheduler underestimates process requirements

OS Support for Processes

- Processes ask for services from the OS using **system calls**
  - **trap** instructions launch interrupt service routines

System Calls for Process Management

- **Creation**
  - a “parent” process spawns a “child” process; a **fork** in UNIX
    - child may or may not inherit parent’s memory
    - child is added to the ready queue
  - the parent-child association is maintained via process IDs (PIDs)

- **Termination**
  - normal: a process asks the OS to delete it; an **exit** in UNIX
    - all resources of a terminated process are deallocated and reclaimed
    - on termination, the child’s PID and output may be passed back to the parent
  - abnormal: another process (typically the parent) can cause termination
    - if the child exceeds its usage, becomes obsolete, or the parent is exiting the system due to some other problem
    - a process (almost always) terminates when its parent does

- **Communication**: Lecture 5/6
- **Coordination**: Lectures 6, 9-11
Example: Process Creation in UNIX

Two system calls: `fork`, `exec`

```c
if ( fork() ) {
} else {
    exec(...)
}
```

```c
if ( fork() ) {
} else {
    exec(...)
}
```

2/4/2004

UNIX System Initialization

![Diagram of UNIX System Initialization]

Outline

- Announcements
  - Please sign up for Lab 1, Part (b) demos
  - Demos on February 9th, 10th
  - Questions?
- Processes (cont’d)
  - process scheduling and other operations
- Threads
  - What are they
  - Multithreading models
- Process Cooperation
  - Why required
  - Shared memory and message passing
  - Introduction to the critical section problem

[Silberschatz/Galvin/Gagne: Sections 4.1-4.5, 5.1-5.5]

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
Threads versus Processes (contd.)

<table>
<thead>
<tr>
<th>a thread</th>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>a traditional process</td>
<td>STACK</td>
</tr>
<tr>
<td></td>
<td>DATA</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>a multi-threaded process</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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
    - operations are faster (context switch, communication, control)
    - 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, XP (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
  - Please sign up for Lab 1, Part (b) demos
  - Demos on February 9th, 10th
  - Questions?
- Processes (cont’d)
  - Process scheduling and other operations
- Threads
  - What are they
  - Multithreading models
- Process Cooperation
  - Why required
  - Shared memory and message passing
  - Introduction to the critical section problem

[Silberschatz/Galvin/Gagne: Sections 4.1-4.5, 5.1-5.5]
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

  \[
  \begin{align*}
  N & : \text{integer} \quad \text{-- buffer size} \\
  \text{nextin} = \text{nextout} = 1 \text{ initially;} & \quad \text{-- start of buffer} \\
  \text{buffer: array of size } N \\
  \text{Producer:} & \\
  & \text{Repeat} \\
  & \quad \text{-- produce an item in tempin} \\
  & \quad \text{while } (\text{nextin+1}) \mod n = \text{nextout do wait-a-bit}; \\
  & \quad \text{buffer}[\text{nextin}] := \text{tempin}; \\
  & \quad \text{nextin := (nextin+1) mod n;}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Consumer:} & \\
  & \text{Repeat} \\
  & \quad \text{while } \text{nextin} = \text{nextout do wait-a-bit;} \\
  & \quad \text{tempout := buffer}[\text{nextout}]; \\
  & \quad \text{nextout := (nextout+1) mod n;} \\
  & \quad \text{-- consume the item in tempout}
  \end{align*}
  \]

Bounded Buffers Using Counters

  \[
  \begin{align*}
  N & : \text{integer} \quad \text{-- buffer size} \\
  \text{counter: integer} = 0 \text{ initially;} & \quad \text{-- start of buffer} \\
  \text{nextin} = \text{nextout} = 1 \text{ initially;} \\
  \text{buffer: array of size } N \\
  \text{Producer:} & \\
  & \text{Repeat} \\
  & \quad \text{-- produce an item in tempin} \\
  & \quad \text{while } \text{counter} = N \text{ do wait-a-bit;} \\
  & \quad \text{buffer}[\text{nextin}] := \text{tempin}; \\
  & \quad \text{nextin := (nextin+1) mod n;} \\
  & \quad \text{counter := counter+1;}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Consumer:} & \\
  & \text{Repeat} \\
  & \quad \text{while } \text{counter} = 0 \text{ do wait-a-bit;} \\
  & \quad \text{tempout := buffer}[\text{nextout}]; \\
  & \quad \text{nextout := (nextout+1) mod n;} \\
  & \quad \text{counter := counter-1;} \\
  & \quad \text{-- consume the item in tempout}
  \end{align*}
  \]

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

Interleaving of Increment/Decrement

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

  \[
  \begin{align*}
  \text{Producer} & \\
  & \text{register1 := counter} \\
  & \text{register1 := register1 + 1} \\
  & \text{counter := register1}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Consumer} & \\
  & \text{register2 := counter} \\
  & \text{register2 := register2 - 1} \\
  & \text{counter := register2}
  \end{align*}
  \]

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

  \[
  \begin{align*}
  \text{Producer} & \\
  & \text{register1 := counter} \\
  & \text{register1 := register1 + 1} \\
  & \text{counter := register1}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Consumer} & \\
  & \text{register2 := counter} \\
  & \text{register2 := register2 - 1} \\
  & \text{counter := register2}
  \end{align*}
  \]
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