(Review) Scheduling Processes

- A process is a program in execution
- A multiprogramming OS simultaneously supports multiple processes

- OS decides which process to run next
  - multiplex CPU among ready processes
  - swap out a process in a time-shared system
  - start and stop processes for accessing secondary memory and I/O

- Three main concerns
  - what happens to the process currently using the CPU?
  - how do you keep track of what each process should be doing?
  - how do you decide which process does what?

(Review) Concern 1: Process Context Switch

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

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

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: Later in this lecture
- Coordination: Lectures 5, 8-11
Example: Process Creation in UNIX

Two system calls: fork, exec

before fork()

if ( fork() )
{
    // parent resumes here
}
else{
    exec(…)
    // child resumes here
}

after fork()

if ( fork() )
{
    // child resumes here
}
else{
    exec(…)
    // parent resumes here
}

after exec()

UNIX System Initialization

bootstrap

swapper

wait until init exits

system shutdown

process 0

fork

process 1

wait until all children exit

init

fork

getty

exec

login

wait until all

children exit

as many as

available

terminals

exec

SHELL

exit

user commands

user environment

Outline

* Announcements
  * Lab 1 due Monday, Feb 11, 2002
  * Instructions about submission process/demos will be posted on mailing list

* Processes
  * Process scheduling
  * Operations on processes

* Threads
  * What are they
  * Multithreading models
  * Examples

* Process Cooperation
  * Why required
  * Shared memory and message passing
  * Synchronization

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.)

- A thread: each thread has its own stack and local data.
- A traditional process: all threads share the same stack and data.

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 (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
    - 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
  - Lab 1 due Monday, Feb 11, 2002
    - Instructions about submission process/demos will be posted on mailing list
- Processes
  - Process scheduling
  - Operations on processes
- Threads
  - What are they
  - Multithreading models
  - Examples
- Process Cooperation
  - Why required
  - Shared memory and message passing
  - Synchronization

[Silberschatz/Galvin/Gagne: Sections 4.2-4.5, Chapter 5, 7.1]
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

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

Bounded Buffers Using Counters

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

Interleaving of Increment/Decrement

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

\[
\begin{align*}
\text{Producer} & \\
\quad \text{register}1 := \text{counter} \\
\quad \text{register}1 := \text{register}1 + 1 \\
\quad \text{counter} := \text{register}1 \\
\text{Consumer} & \\
\quad \text{register}2 := \text{counter} \\
\quad \text{register}2 := \text{register}2 - 1 \\
\quad \text{counter} := \text{register}2
\end{align*}
\]

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

\[
\begin{align*}
\text{Producer} & \\
\quad \text{register}1 := \text{counter} \\
\quad \text{register}1 := \text{register}1 + 1 \\
\quad \text{counter} := \text{register}1 \\
\text{Consumer} & \\
\quad \text{register}2 := \text{counter} \\
\quad \text{register}2 := \text{register}2 - 1 \\
\quad \text{counter} := \text{register}2
\end{align*}
\]

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
    - Send\text{Message}[channel, body] returns id
    - Await\text{Reply}[id]
    - Recv\text{Message}[port] returns id
    - Send\text{Reply}[id, body]
- Many variants: See Section 4.5
  - Focus on shared memory for next few lectures
The Problem

- Increment and decrement are not \textit{atomic} or \textit{uninterruptable}
  - two or more operations are executed \textit{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 \textit{atomic}
    - byte read; byte write;
    - word read; word write

- The code containing these operations creates a \textit{race condition}
  - produces inconsistencies in shared data

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

The Solution

- The producer and consumer processes need to \textit{synchronize}
  - so that they \textit{do not} access shared variables at the same time

- this is called \textit{mutual exclusion}
  - the \textit{shared} and \textit{critical} variables can be accessed by only one process at a time
  - access must be \textit{serialized} even if the processes attempt \textit{concurrent} access
    - in the previous example: counter increment and decrement operations

- General framework for achieving this: \textit{Critical Sections}
  - work independent of the particular context or need for synchronization