Outline

• Announcements
  – Lab 1 due Wednesday, Jan 31, 2001
  – Instructions about submission process/demos in mailing list
  – Questions?

• Processes
  – process scheduling
  – operations on processes
  – Threads

• Process Cooperation
  – why required
  – shared memory and message passing
  – synchronization

[ Silberschatz/Galvin: Sections 4.5-4.6, 6.1-6.2]

(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?

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 enter the
  - new state
  - ready state
  - running state
  - waiting state
  - terminated state

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 (PIPs)

- 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`

```
if ( fork() ) {
} else {
  exec(…)
}
```

before fork()

```
if ( fork() ) {
} else {
  exec(…)
}
```

parent resumes here

child resumes here

after fork()

```
if ( fork() ) {
} else {
  exec(…)
}
```

child resumes here

after exec()

UNIX System Initialization

```
bootstrap
```  
```
swapper
```  
```
prompt
```  
```
init
```  
```
wait until init exits
```  
```
process 0
```  
```
wait until all children exit
```  
```
process 1
```  
```
wait until all children exit
```  
```
init
```  
```
wait until all children exit
```  
```
system shutdown
```  
```
getty
```  
```
wait until all terminals exit
```  
```
login
```  
```
wait until all terminals exit
```  
```
exec
```  
```
wait until all terminals exit
```  
```
SHELL
```  
```
wait until all terminals exit
```  
```
exit
```  
```
user commands
```  
```
wait until all terminals exit
```  
```
exit
```  
```
user environment
```  
```
exit
```  
```
... 
```  
```
exit
```  
```
user environment
```  
```
exit
```  
```
user environment
```  
```
exit
```  
```
user environment
```  
```
exit
```  
```
user environment
```  

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.)
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

• Benefits for the OS
  – context switching is more efficient
    • memory mappings can remain unchanged
    • cache need not be flushed

Types of Threads

• User-level threads
  – 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
  – 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

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 KT with no LWP
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

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 4.6
  - Focus on shared memory for next few lectures

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
  
  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

Bounded Buffers Using Counters

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
Interleaving of Increment/Decrement

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

  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
  counter := register1

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

The Problem

- Increment and decrement are not atomic or uninterruptible
  - 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