Outline

- Announcements
  - Lab 1 due Wednesday, Feb 02, 2000
  - Lab 1 demos on Thursday, Feb 03, 2000
  - Questions?

- Processes
  - process scheduling
  - operations on processes
  - Threads

- Process Cooperation
  - why required
  - shared memory and message passing
  - synchronization

[Silberschatz/Galvin: Chapter 4]

(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
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()) {
    
} else {
    exec(...)
}

child resumes here

after fork()

parent resumes here

after exec()

UNIX System Initialization

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's with no LWP

Outline

- Announcements
  - Lab 1 due Wednesday, Feb 02, 2000
  - Lab 1 demos on Thursday, Feb 03, 2000
  - Questions?

- Processes
  - process scheduling
  - operations on processes
  - Threads

- Process Cooperation
  - why required
  - shared memory and message passing
  - synchronization

[ Silberschatz/Galvin: Chapter 4]
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
  
  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]
    - SendReply[id, body]

- Many variants: See Section 4.6
  - Focus on shared memory for next few lectures

Bounded Buffers Using Counters

N: integer                        -- buffer size
counter: integer
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;

**Consumer:**

Repeat
while counter = 0 do wait-a-bit;
tempout := buffer[nextout];
nextout := (nextout+1) mod n;

Producer and Consumer processes are asynchronous! execution of these two statements can be interleaved (e.g., because of interrupts)

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

Producer
- \( \text{register1} \leftarrow \text{counter} \)
- \( \text{register1} \leftarrow \text{register1} + 1 \)
- \( \text{counter} \leftarrow \text{register1} \)

Consumer
- \( \text{register2} \leftarrow \text{counter} \)
- \( \text{register2} \leftarrow \text{register2} - 1 \)
- \( \text{counter} \leftarrow \text{register2} \)

- An interleaving
  - \( \text{counter} = 5 \); a producer followed by a consumer

Producer
- \( \text{register1} \leftarrow \text{counter} \) \( \text{[register1 = 5]} \)
- \( \text{register1} \leftarrow \text{register1} + 1 \) \( \text{[register1 = 6]} \)
- \( \text{counter} \leftarrow \text{register1} \) \( \text{[counter = 6]} \)

Consumer
- \( \text{register2} \leftarrow \text{counter} \) \( \text{[register2 = 5]} \)
- \( \text{register2} \leftarrow \text{register2} - 1 \) \( \text{[register2 = 4]} \)
- \( \text{counter} \leftarrow \text{register2} \) \( \text{[counter = 4]} \)

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 are written as

  ENTRY-SECTION
  CRITICAL-SECTION-CODE
  EXIT-SECTION

  - the ENTRY-SECTION controls access to make sure 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?
  - build on top of atomic memory load/store operations
  - provide higher-level primitives: Next Lecture
Two-Process Solutions: Turn Counters

- Shared integer variable: \texttt{turn} (initialized to 0)
  - for processes \texttt{P}_i and \texttt{P}_j, \texttt{P}_i executes:
    \begin{verbatim}
    while (turn \neq i) wait-a-bit;
    CRITICAL SECTION;
    turn := j;
    \end{verbatim}
    - the while loop is the \textit{entry} section
      - process \texttt{P}_i waits till its turn occurs
    - the single instruction \texttt{turn := j} constitutes the \textit{exit} section
      - informs the other process of its turn

- Mutual exclusion?
  - assume atomic loads and stores

- Drawbacks?
  - if \texttt{P}_j never wants to execute the critical section, \texttt{P}_i cannot reenter;
    - access \textit{must} alternate

Two-Process Solutions: Array of Flags

- Boolean array: \texttt{flag} (initialized to false)
  - \texttt{P}_i executes:
    \begin{verbatim}
    1: flag[i] := true;
    2: while flag[j] wait-a-bit;
    3: flag[i] := false;
    \end{verbatim}

- Mutual exclusion?
  - suppose \textit{not}; then \exists a sequence s.t. both \texttt{P}_i and \texttt{P}_j are in the critical section
  - when \texttt{P}_i entered the critical section,
    - it found \texttt{flag[j]} to be false at time \( t_1 \): instruction 2
    - it had set \texttt{flag[i]} to true at time \( t_0 \) \((0 < t_1)\)
  - if \texttt{P}_j was already in the critical section
    - it must have found \texttt{flag[i]} to be false at time \( t_3 \) \(( t_3 < t_0 )\): instruction 2
      - it must have set \texttt{flag[j]} to true at time \( t_2 \) \(( t_2 < t_3)\)
    - so \texttt{P}_i cannot find \texttt{flag[j]} false at time \( t_1 > t_0 > t_2 \)!

Is This Good Enough?

- \textbf{No}: \texttt{P}_i and \texttt{P}_j can be looping on instruction 2 forever
  - both \texttt{P}_i and \texttt{P}_j set their flags to true and wait for the other flag to be false
  - leads to deadlock, which will be discussed in detail later

- So, what criteria should a general critical-section solution satisfy?

Criteria for Correctness

Three conditions
- \textbf{Mutual exclusion}
- \textbf{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
- \textbf{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
  – process \( P_i \) executes
    1: \( \text{flag}[i] := \text{true} \)
    2: \( \text{turn} := j \)
    3: while \( (\text{flag}[j] \text{ and } (\text{turn} == j)) \) wait-a-bit
    CRITICAL SECTION
    4: \( \text{flag}[i] := \text{false} \)

• Does the algorithm satisfy the three criteria?

Petersen’s Algorithm: Mutual Exclusion

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

• Suppose: \( P_i \) is in its critical section, and \( P_j \) is wanting to enter
• This can happen only if either
  – (case 1) \( P_i \) found \( \text{flag}[j] \) false, or
  – (case 2) \( P_i \) found \( \text{turn} == i \)
  – in the first case: \( P_j \) will set \( \text{turn} \) after \( P_i \) did, and find \( \text{turn} == i \)
  – in the second case: \( P_j \) has already set \( \text{turn} = i \)
  – in both cases: \( P_j \) will wait till \( \text{flag}[i] == \text{false} \)

Petersen’s Algorithm: Progress and Bounded Waiting

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

• To prove progress:
  – if \( P_i \) is not ready to enter the critical section
    • \( \text{flag}[j] \) will be false ★ \( P_i \) can enter

• To prove bounded waiting:
  – let \( P_i \) be in the critical section and \( P_j \) be waiting on instruction 3 above
  – if \( P_i \) exits and goes elsewhere,
    • either \( P_i \) will find \( \text{flag}[i] \) to be false
    • if not, \( P_i \) will attempt to reenter the critical section, setting \( \text{turn} := j \)
  – in either case, \( P_i \) 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 6.2.2)
  – 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 (Next Lecture)
  – locks (mutexes), semaphores, condition variables
  – rely on more support from hardware
    • disabling of interrupts
    • atomic read-modify-write operations
N-Process Synchronization

**Intuition:**
- Processes ask for a *ticket* from an *agent* and get an integer valued ticket
- Processes are not guaranteed to receive unique tickets
- A process waits until all processes with smaller ticket values have finished going through the critical region
- In case of a tie, let the process with the smaller PID go first
- Leads to a FCFS prioritizing strategy

The algorithm is akin to taking a ticket and waiting for a turn in a bakery and is called the *bakery algorithm*

**The Bakery Algorithm**

```plaintext
1: choosing[0] := true
2: number[0] := max(number[0], number[1], ..., number[n-1]) + 1
3: choosing[0] := false
4: for j := 0 to n-1
5: do begin
6: while choosing[j] do wait-a-bit
7: while number[j] != 0 and (number[j],j) < (number[0],0) do waitabit
8: end
9: number[0] := 0;
```
Bakery Algorithm: Mutual Exclusion

1: choosing[i] := true
2: number[i] := max{number[0], number[1], ..., number[n-1]} + 1
3: choosing[i] := false
4: for j := 0 to n-1
5:   do begin
6:     while choosing[j] do wait-a-bit
7:     while number[j] := 0 and
8:       (number[j],j) < (number[i],i) do wait-a-bit
9:   end

Consider P_i in its critical section, and P_k trying to enter its own
- 3 cases when P_i executes step 2:
  - case 1: P_i found choosing[k] true
  - case 2: P_i found choosing[k] false because P_k had executed step 3.
  - case 3: P_i found choosing[k] false because P_k had not executed step 1.
- In cases 1 and 2, P_i would have found ticket[j] non-zero
  - so it must have found (number[i],i) < (number[k],k)
- In case 3: P_k would compute ticket[k] > ticket[i]
- so, P_k will remain stuck on step 7 while P_i is in the critical section

Bakery Algorithm: Progress and Bounded Waiting

- Assumption: No processes fail
- Ticketed processes will exit in FCFS order
- The set of processes is finite
  - implies that every process will get its turn in finite time