Outline

- Announcements
  - Lab 2 due on February 23rd
    - Demos will be on 23rd (Monday) and 24th (Tuesday)
    - Defer extra credit part (priority scheduler) to Lab 3
  - Questions?

- Process Cooperation (cont’d)
  - Synchronization primitives in real OSes
  - Classical synchronization problems

[Silberschatz/Galvin/Gagne, Sections 7.2 – 7.5]

(Review) Synchronization Primitives

- Locks: A boolean variable $L$, either AVAILABLE or BUSY
  - \textbf{LOCK:} while ( $L$ != AVAILABLE ) wait-a-bit $L = BUSY$;
  - \textbf{UNLOCK:} $L = AVAILABLE$; wake up a waiting process (if any)

- Semaphores: A single integer variable $S$
  - \textbf{WAIT:} while $S \leq 0$ do wait-a-bit; $S := S-1$;
  - \textbf{SIGNAL:} $S := S+1$; wake up a waiting process (if any)

- Condition variables: A process queue (accessed within a critical section)
  - \textbf{WAIT:} associate self with the implicit queue suspend self
  - \textbf{SIGNAL:} wake up one suspended process on queue has no effect if there are no suspended processes
  - \textbf{BROADCAST:} wake up all suspended processes

Synchronization Primitives in Real OSes

- Unix: Single CPU OS
  - implement critical sections using interrupt elevation
    - disallow interrupts that can modify the same data
  - another possibility: interrupts never “force” a context switch
    - they just set flags, or wake up processes
  - primitives
    - \textbf{sleep} (address);
    - \textbf{wake_up} (address); -- wakes up all processes sleeping on address
  - typical code
    \begin{verbatim}
    ENTRY: while (locked) sleep(bufaddr);
    locked = true;
    EXIT: locked = false; wake_up (bufaddr);
    \end{verbatim}
Synchronization Primitives in Real OSes (contd.)

- Solaris 2: multi-CPU OS
  - for brief accesses
    - adaptive mutexes
    - starts off as a standard spinlock
      - if lock is held by running thread, continues to spin
      - valid only on a multi-CPU system
    - otherwise blocks
  - for long-held locks
    - condition variables
    - wait and signal
    - reader-writer locks
      - for frequent mostly read-only accesses

Classical Synchronization Problems

- Commonly encountered problems in operating systems
  - used to test any proposal for a new synchronization primitive

1. Mutual exclusion
  - only one process executes a piece of code (critical section) at any time
  - OS examples: access to shared resources
    - e.g., a printer

2. Sequencing
  - a process waits for another process to finish executing some code
  - OS examples: waiting for an event
    - e.g., recv suspends until there is some data to read on the network

Classical Synchronization Problems (cont’d)

3. Bounded-buffer (also referred to as the Producer-Consumer problem)
  - a pool of n buffers
  - producer process(es) put items into the pool
  - consumer process(es) take items out of the pool
  - issues: mutual exclusion, empty pool, and full pool
  - OS examples: buffering for pipes, file caches, etc.

4. Readers-Writers
  - multiple processes access a shared data object X
    - any number of readers can access X at the same time
    - no writer can access it at the same time as a reader or another writer
  - mutual exclusion is too constraining: WHY?
  - variations:
    - reader-priority: a reader must not wait for a writer
    - writer-priority: a writer must not wait for a reader
  - OS examples: file locks

Classical Synchronization Problems (contd.)

5. Dining Philosophers
  - 5 philosophers
  - 5 chopsticks placed between them
    - to eat requires two chopsticks
  - philosophers alternate between thinking and eating
  - issues: deadlock, starvation, fairness
  - OS examples: simultaneous use of multiple resources
    - e.g., disk bandwidth and storage
Mutual Exclusion and Sequencing Using Semaphores

- Mutual exclusion: Semaphore initialized to 1
  \[ P(S); \]
  \[ \text{CRITICAL SECTION} \]
  \[ V(S); \]

- Sequencing: Semaphore initialized to 0
  \[ \text{process 1 process 2} \]
  \[ B(); \]
  \[ V(S); \]
  \[ P(S); \]
  \[ A(); \]

Bounded-buffer Using Semaphores

- Three semaphores
  - \textit{mutex}: provide mutual exclusion between processes (initial value = 1)
  - \textit{empty}: count the number of empty slots (initial value = N)
  - \textit{full}: count the number of full slots (initial value = 0)

Producer(s):

\[ \text{repeat} \]
\[  \quad // \text{produce item in nextp} \]
\[  \quad P(\text{empty}); \]
\[  \quad P(\text{mutex}); \]
\[  \quad // \text{add nextp to buffer} \]
\[  \quad V(\text{mutex}); \]
\[  \quad V(\text{full}); \]
\[  \quad \text{until false;} \]

Consumer(s):

\[ \text{repeat} \]
\[  \quad P(\text{full}); \]
\[  \quad P(\text{mutex}); \]
\[  \quad // \text{remove item to nextc} \]
\[  \quad V(\text{mutex}); \]
\[  \quad V(\text{empty}); \]
\[  \quad // \text{consume item in nextc} \]
\[  \quad \text{until false;} \]

Readers-Writers Using Semaphores

To allow multiple readers, synchronize only the first/last reader with writers

Reader(s)

\[ P(x); \]
\[ \quad \text{rcount} := \text{rcount} + 1; \]
\[ \quad \text{if (rcount == 1) then } P(wsem); \]
\[ \quad V(x); \]
\[ \text{READ} \]
\[ \quad \text{can release either waiting readers or writers} \]
\[ P(x); \]
\[ \quad \text{rcount} := \text{rcount} - 1; \]
\[ \quad \text{if (rcount == 0) then } V(wsem); \]
\[ \quad V(x); \]

Writer(s)

\[ P(wsem); \]
\[ \text{WRITE} \]
\[ \text{can release either waiting readers or writers} \]

Readers-Writers Using Semaphores: Writer-Priority

Have a writer block out subsequent readers (same as readers block out writers)

Reader

\[ P(rsem); \]
\[ \quad P(x); \]
\[ \quad \text{rcount} := \text{rcount} + 1; \]
\[ \quad \text{if (rcount == 1) then } P(wsem); \]
\[ \quad V(x); \]
\[ \quad V(rsem); \]
\[ \text{READ} \]
\[ \quad \text{writers can queue up preventing a waiting writer from setting rsem} \]

Writer

\[ P(y); \]
\[ \quad wcount := wcount + 1; \]
\[ \quad \text{if (wcount == 1) then } P(rsem); \]
\[ \quad V(y); \]
\[ \quad P(wsem); \]
\[ \text{WRITE} \]
\[ \quad V(wsem); \]
\[ \quad P(y); \]
\[ \quad \text{can release either waiting readers or writers} \]
\[ \quad wcount := wcount - 1; \]
\[ \quad \text{if (wcount == 0) then } V(rsem); \]
\[ \quad V(y); \]
Readers-Writers Using Semaphores: Writer-Priority (2)

Reader

\[
P(z); \quad P(rsem); \quad P(x); \quad P(wsem); \quad V(rsem); \quad V(z);
\]
\[
rcount := rcount + 1; \quad \text{if} \ (rcount == 1) \ \text{then} \ P(wsem); \quad V(x); \quad V(rsem); \quad V(z);
\]

Reader

\[
P(x); \quad P(wsem); \quad V(x); \quad V(wsem); \quad P(y); \quad wcount := wcount + 1; \quad \text{if} \ (wcount == 1) \ \text{then} \ P(rsem); \quad V(y); \quad P(rsem); \quad V(x);
\]

Writer

\[
P(y); \quad wcount := wcount - 1; \quad \text{if} \ (wcount == 0) \ \text{then} \ V(rsem); \quad V(y);
\]

Dining Philosophers Using Semaphores

Philosopher

\[
P(\text{chopstick}[i]); \quad P(\text{chopstick}[i+1 \mod 5]); \quad EAT \quad V(\text{chopstick}[i]); \quad V(\text{chopstick}[i+1 \mod 5]); \quad \text{THINK}
\]

Philosopher

\[
P(\text{chopstick}[j]); \quad P(\text{chopstick}[j+1 \mod 5]); \quad EAT \quad V(\text{chopstick}[j]); \quad V(\text{chopstick}[j+1 \mod 5]); \quad \text{THINK}
\]

Dining Philosophers Using Semaphores - 2

• Deadlock

\[
a \text{ set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set}
\]

• details in Lectures 10 and 11.

Philosopher (even i)

\[
P(\text{chopstick}[i]); \quad P(\text{chopstick}[i+1 \mod 5]); \quad EAT \quad V(\text{chopstick}[i]); \quad V(\text{chopstick}[i+1 \mod 5]); \quad \text{THINK}
\]

Philosopher (odd i)

\[
P(\text{chopstick}[i+1 \mod 5]); \quad P(\text{chopstick}[i]); \quad EAT \quad V(\text{chopstick}[i+1 \mod 5]); \quad V(\text{chopstick}[i]); \quad \text{THINK}
\]

A Larger Example: A Barbershop Problem

• Example taken from

\[
\text{Operating Systems: Internals and Design Principles, 3rd Edition}
\]


• The problem: Orchestrating activities in a barbershop

\[
- \text{3 chairs, 3 barbers, 1 cash register, waiting area: 4 customers on a sofa, plus additional standing room}
- \text{Fire codes limit total number of customers to 20 at a time}
- \text{A customer}
\]

\[
- \text{Will not enter the shop if it is filled to capacity}
- \text{Takes a seat on the sofa, or stands if sofa is filled}
- \text{When a barber is free, the customer waiting longest on sofa is served}
- \text{The customer standing the longest takes up seat on the sofa}
- \text{When a customer’s haircut is finished, any barber can accept payment but because of the single cash register, only one payment is accepted at a time}
- \text{Barbers divide their time between cutting hair, accepting payment, and sleeping}
\]

• Alternate solutions

\[
- \text{allow at most 4 philosophers to sit simultaneously at the table}
- \text{allow a philosopher to pick up chopsticks only if both are available}
\]

• All of these solutions suffer from the possibility of starvation!
A Barbershop Problem (cont’d)

- Shop and sofa capacity
  - max_capacity (initial value = 20)
  - sofa (initial value = 4)
- Barber chair capacity
  - barber_chair (initial value = 3)
- Ensuring customers are in barber chair
  - cust_ready (initial value = 0)
  - finished (initial value = 0)
  - leave_b_chair (initial value = 0)
- Paying and receiving
  - payment (initial value = 0)
  - receipt (initial value = 0)
- Coordinating barber functions
  - coord (initial value = 0)

Customer
- P( max_capacity );
- P( sofa );
- P( barber_chair );
- P( leave_b_chair );
- P( receipt );
- V( payment );
- V( receipt );
- V( max_capacity );

Barber
- P( cust_ready );
- P( coord );
- V( coord );
- V( finished );
- V( barber_chair );
- V( leave_b_chair );
- V( max_capacity );

Cashier
- P( payment );
- P( coord );
- V( payment );
- V( receipt );
- V( max_capacity );

A Barbershop Problem (cont’d): Mutual Exclusion

Customer
- P( max_capacity );
- P( sofa );
- P( barber_chair );
- P( leave_b_chair );
- P( receipt );
- V( payment );
- V( receipt );
- V( max_capacity );

Barber
- P( cust_ready );
- P( coord );
- V( coord );
- V( finished );
- V( barber_chair );
- V( leave_b_chair );
- V( max_capacity );

Cashier
- P( payment );
- P( coord );
- V( payment );
- V( receipt );
- V( max_capacity );

A Barbershop Problem (cont’d): Bounded Buffer

Customer
- P( max_capacity );
- P( sofa );
- P( barber_chair );
- P( leave_b_chair );
- P( receipt );
- V( payment );
- V( receipt );
- V( max_capacity );

Barber
- P( cust_ready );
- P( coord );
- V( coord );
- V( finished );
- V( barber_chair );
- V( leave_b_chair );
- V( max_capacity );

Cashier
- P( payment );
- P( coord );
- V( payment );
- V( receipt );
- V( max_capacity );
A Barbershop Problem (cont’d): Sequencing

- Shop and sofa capacity
  - max_capacity (:= 20)
  - sofa (:= 4)
- Barber chair capacity
  - barber_chair (:= 3)
- Ensuring customers are in barber chair
  - cust_ready (:= 0)
  - finished (:= 0)
  - leave_b_chair (:= 0)
- Paying and receiving
  - payment (:= 0)
  - receipt (:= 0)
- Coordinating barber functions
  - coord (:= 0)

Customer
P( max_capacity );
// enter shop
P( sofa );
// sit on sofa
P( barber_chair );
// get up from sofa
V( sofa );
// sit in barber chair
V( cust_ready );
// leave barber chair
P( finished );
// leave barber chair
V( leave_b_chair );
// pay
V( payment );
// exit shop
V( max_capacity );

Barber
P( cust_ready );
// cut hair
P( coord );
// wait for customer to leave
V( coord );
V( finished );
// tell next customer to hop on
V( barber_chair );

Cashier
P( payment );
// accept payment
P( coord );
// receive receipt
V( coord );
V( receipt );

• Some problems with the current solution
  - since all customers are waiting on the same semaphore (finished), the one who started earliest is released when a barber does V(finished)
    - even if the haircut is not done
  - similar problem with the cashier and the pay and receipt semaphores
    - cashier may accept money from one customer and release another
    - a customer needs to wait on the sofa even if a barber chair is free
• All of these can be solved using additional semaphores