Outline

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

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

\[\text{Silberschatz/Galvin/Gagne, Sections 7.2 – 7.5}\]
(Review) Synchronization Primitives

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

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

- **Condition variables**: A process queue (accessed within a critical section)
  - **WAIT**: associate self with the implicit queue
    suspend self
  - **SIGNAL**: wake up one suspended process on queue
    has no effect if there are no suspended processes
  - **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
    - `sleep` (address);
    - `wake_up` (address); -- wakes up all processes sleeping on address
  - typical code
    ```c
    ENTRY: while (locked) sleep(bufaddr);
            locked = true;
    EXIT:  locked = false; wake_up (bufaddr);
    ```
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
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
  \[ \text{\texttt{P(S);}} \]
  \[ \text{\texttt{CRITICAL SECTION}} \]
  \[ \text{\texttt{V(S);}} \]

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

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

**Producer(s):**

```plaintext
repeat
  // produce item in nextp
  P( empty );
  P( mutex );
  // add nextp to buffer
  V( mutex );
  V( full );
until false;
```

**Consumer(s):**

```plaintext
repeat
  P( full );
  P( mutex );
  // remove item to nextc
  V( mutex );
  V( empty );
  // consume item in nextc
until false;
```
Readers- Writers Using Semaphores

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

Reader(s)

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

READ

\[
P(x);
\]
\[
rcount := rcount - 1;
\]
\[
\text{if (rcount == 0) then } V(wsem);\]
\[
V(x);
\]

Writer(s)

\[
P(wsem);\]

\[
\text{WRITE}\]
\[
V(wsem);\]

stream of readers can starve writers

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**

\[ P(rsem); \]
\[ P(x); \]
\[ rcount := rcount + 1; \]
\[ if (rcount == 1) then P(wsem); \]
\[ V(x); \]
\[ V(rsem); \]

READ

\[ P(x); \]
\[ rcount := rcount - 1; \]
\[ if (rcount == 0) then V(wsem); \]
\[ V(x); \]

**Writer**

\[ P(y); \]
\[ wcount := wcount + 1; \]
\[ if (wcount == 1) then P(rsem); \]
\[ V(y); \]
\[ P(wsem); \]
\[ WRITE \]
\[ V(wsem); \]

\[ P(y); \]
\[ wcount := wcount - 1; \]
\[ if (wcount == 0) then V(rsem); \]
\[ V(y); \]
Readers-Writers Using Semaphores: Writer-Priority (2)

**Reader**

\[
\begin{align*}
&P(z); \\
&P(rsem); \\
&P(x); \\
&\text{rcount} := \text{rcount} + 1; \\
&\quad \text{if (rcount == 1) then } P(wsem); \\
&V(x); \\
&V(rsem); \\
&V(z); \\
&\text{READ} \\
&P(x); \\
&\quad \text{rcount} := \text{rcount} - 1; \\
&\quad \text{if (rcount == 0) then } V(wsem); \\
&V(x); \\
\end{align*}
\]

**Writer**

\[
\begin{align*}
&P(y); \\
&wcount := wcount + 1; \\
&\quad \text{if (wcount == 1) then } P(rsem); \\
&V(y); \\
&P(wsem); \\
&\quad \text{WRITE} \\
&V(wsem); \\
&P(y); \\
&\quad wcount := wcount - 1; \\
&\quad \text{if (wcount == 0) then } V(rsem); \\
&V(y); \\
\end{align*}
\]
Dining Philosophers Using Semaphores

Philosopher\(_i\)

\[
\begin{align*}
&\text{P( chopstick[i] );} \\
&\text{P( chopstick[i+1 mod 5] );} \\
&\text{EAT} \\
&\text{V( chopstick[i] );} \\
&\text{V( chopstick[i+1 mod 5] );} \\
&\text{THINK}
\end{align*}
\]

Philosopher\(_{(j=i+1 \ mod \ 5)}\)

\[
\begin{align*}
&\text{P( chopstick[j] );} \\
&\text{P( chopstick[j+1 mod 5] );} \\
&\text{EAT} \\
&\text{V( chopstick[j] );} \\
&\text{V( chopstick[j+1 mod 5] );} \\
&\text{THINK}
\end{align*}
\]

- **Deadlock**
  
  *a 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.
Dining Philosophers Using Semaphores - 2

- **Alternate solutions**
  - allow at most 4 philosophers to sit simultaneously at the table
  - allow a philosopher to pick up chopsticks only if both are available

- All of these solutions suffer from the possibility of *starvation*!
A Larger Example: A Barbershop Problem

• Example taken from

• The problem: Orchestrating activities in a barbershop
  – 3 chairs, 3 barbers, 1 cash register,
    waiting area: 4 customers on a sofa, plus additional standing room
  – Fire codes limit total number of customers to 20 at a time
  – A customer
    • Will not enter the shop if it is filled to capacity
    • Takes a seat on the sofa, or stands if sofa is filled
    • When a barber is free, the customer waiting longest on sofa is served
      The customer standing the longest takes up seat on the sofa
    • 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
    • Barbers divide their time between cutting hair, accepting payment, and
      sleeping
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)
    - barber waits for customer
  - `finished` (initial value = 0)
    - customer waits for haircut to finish
  - `leave_b_chair` (initial value = 0)
    - barber waits for chair to empty

- Paying and receiving
  - `payment` (initial value = 0)
    - cashier waits for customer to pay
  - `receipt` (initial value = 0)
    - customer waits for cashier to ack

- Coordinating barber functions
  - `coord` (initial value = 0)
    - wait for a barber resource to free up
A Barbershop Problem (cont’d)

- **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**

```plaintext
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 );
P( finished );
// leave barber chair
V( leave_b_chair );
// leave shop
V( max_capacity );
```

**Barber**

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

**Cashier**

```plaintext
P( payment );
P( coord );
// accept payment
V( coord );
V( receipt );
```
A Barbershop Problem (cont’d): Mutual Exclusion

- 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

\[
P(\text{max\_capacity}); \\
// enter shop \\
P(\text{sofa}); \\
// sit on sofa \\
P(\text{barber\_chair}); \\
// get up from sofa \\
V(\text{sofa}); \\
// sit in barber chair \\
P(\text{cust\_ready}); \\
// get from sofa \\
V(\text{sofa}); \\
// in barber chair \\
P(\text{finished}); \\
// leave barber chair \\
V(\text{leave\_b\_chair}); \\
// pay \\
V(\text{payment}); \\
P(\text{receipt}); \\
// exit shop \\
V(\text{max\_capacity});
\]

Barber

\[
P(\text{cust\_ready}); \\
P(\text{coord}); \\
// cut hair \\
V(\text{coord}); \\
V(\text{finished}); \\
// wait for customer to leave \\
P(\text{leave\_b\_chair}); \\
// tell next customer to hop on \\
V(\text{barber\_chair});
\]

Cashier

\[
P(\text{payment}); \\
P(\text{coord}); \\
// accept payment \\
V(\text{coord}); \\
V(\text{receipt});
\]
A Barbershop Problem (cont’d): Bounded Buffer

- 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 );`
  // wait for customer to leave
- `V( leave_b_chair );`
  // tell next customer to hop on
- `V( max_capacity );`
  // exit shop

**Barber**

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

**Cashier**

- `P( payment );`
- `P( coord );`
  // accept payment
- `V( coord );`
- `V( receipt );`

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)
A Barbershop Problem (cont’d)

- Some problems with the current solution
  - since all customers are waiting on the same semaphore (\textit{finished}), the one who started earliest is released when a barber does \texttt{V(finished)}
    - even if the haircut is not done
  - similar problem with the cashier and the \texttt{pay} and \texttt{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