Frequently asked questions from the previous class survey

- Other real-world examples?
- Critical Sections
  - Parametrized
  - How can wait be bounded when each process is doing its own thing?
  - Entry/Exit section protocol: Responsibility of the programmer?
- What happens in the critical section? Access to shared memory?
- Atomic
- How do hardware-assisted locks deal with priority?
Topics covered in the lecture

- TestAndSet
- Using TestAndSet to satisfy critical section requirements
- Semaphores
- Classical process synchronization problems

TestAndSet()

```java
boolean TestAndSet(boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Sets target to true and returns old value of target
**TestAndSet**: Shared boolean variable `lock` initialized to `false`

```c
do {
    while (TestAndSet(&lock)) {};  // critical section
    lock = FALSE;
}

// remainder section

} while (TRUE);  // To break out: Return value of TestAndSet should be FALSE
```

- **If two TestAndSet() are executed simultaneously**, they will be executed **sequentially** in some arbitrary order.

---

**Using Test-and-Set to Satisfy Critical Section Requirements**
Using `TestAndSet` to satisfy all critical section requirements

- **N processes**

- **Data structures initialized to FALSE**
  - `boolean waiting[n];`
  - `boolean lock;`

  These data structures are maintained in shared memory.

---

The entry section for process i

```c
waiting[i] = TRUE;
key = TRUE;

while (waiting[i] && key) {
    key = TestAndSet(&lock);
}

waiting[i] = FALSE;
```

First process to execute `TestAndSet` will find `key == false`; enter critical section.

Everyone else must wait.

Will break out only if `waiting[i] == FALSE OR key == FALSE`
The exit section: Part I
Finding a suitable waiting process

If a process is not waiting move to the next one

\[ j = (i + 1) \mod n; \]

while \((j \neq i) \land \neg \text{waiting}[j]\) {
  \[ j = (j+1) \mod n \]
}

Will break out at \(j=i\) if there are no waiting processes

If a process is waiting: break out of loop

The exit section: Part II
Finding a suitable waiting process

Could NOT find a suitable waiting process

\[
\begin{align*}
\text{if } (j==i) \{ \\
& \text{lock = FALSE;} \\
\} \text{ else } \{ \\
& \text{waiting}[j] = FALSE;
\}
\end{align*}
\]

Found a suitable waiting process
Mutual exclusion

- The variable $\text{waiting}[i]$ can become \texttt{false} \text{ ONLY if another process leaves its critical section}
  - \textbf{Only one} $\text{waiting}[i]$ is set to \texttt{FALSE}

Progress

- A process exiting the critical section
  1. Sets $\text{lock}$ to \texttt{FALSE}
     OR
  2. $\text{waiting}[j]$ to \texttt{FALSE}

- Allows a process that is \texttt{waiting} to \texttt{proceed}
Bounded waiting requirement

\[
j = (i + 1) \mod n;
\]

\[
\text{while } (j \neq i) \text{ && !waiting}[j]\{ \\
j = (j+1) \mod n \\
\}
\]

□ **Scans** waiting[] in the cyclic ordering

\[
(i+1, i+2, \ldots n, 0, \ldots, i-1)
\]

□ ANY waiting process trying to enter critical section will do so in \((n-1)\) turns
Semaphores

- Semaphore $S$ is an integer variable
- Once initialized, accessed through **atomic** operations
  - `wait()`
  - `signal()`

Modifications to the integer value of semaphore execute indivisibly

```c
wait(S) {
    while (S<=0) {
        ; //no operation
    }
    S--;
}

signal(S) {
    S++;
}
```
Types of semaphores

- **Binary semaphores**
  - The value of $S$ can be 0 or 1
  - Also known as **mutex locks**

- **Counting semaphores**
  - Value of $S$ can range over an **unrestricted domain**

Using the Binary semaphore to deal with the critical section problem

```c
mutex is initialized to 1

{ do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section

} while (TRUE);
```
Suppose we require $S_2$ to execute only after $S_1$ has executed

Semaphore $\text{synch}$ is initialized to 0

Wait for $\text{synch}$ to be $> 0$

$\text{wait(\text{synch});}$

$S_1;$

$\text{signal(\text{synch});}$

$S_2;$

Set $\text{synch}$ to 1

PROCESS $P_1$ PROCESS $P_2$

The counting semaphore

- Controls access to a **finite** set of resource instances
- **INITIALIZED** to the number of resources available
- **Resource Usage**
  - $\text{wait():}$ To *use* a resource
  - $\text{signal():}$ To *release* a resource
- When all resources are being used: $S==0$
  - Block until $S > 0$ to use the resource
Problems with the basic semaphore implementation

- **{C1}** If there is a process in the critical section
- **{C2}** If another process tries to enter its critical section
  - Must loop continuously in entry code
  - **Busy waiting!**
    - Some other process could have used this more productively!
  - Sometimes these locks are called **spinlocks**
    - One advantage: No context switch needed when process must wait on a lock

Overcoming the need to busy wait

- During wait if \( S == 0 \)
  - Instead of busy waiting, the process **blocks** itself
  - Place process in waiting queue for \( S \)
  - Process state switched to **waiting**
  - CPU scheduler picks another process to execute

- **Restart** process when another process does signal
  - Restarted using `wakeup()`
  - Changes process state from **waiting** to **ready**
Defining the semaphore

typedef struct {
    int value;
    struct process *list;
} semaphore;

The `wait()` operation to eliminate busy waiting

```c
wait(semaphore *S){
    S->value--;
    if (S->value < 0) {
        add process to S->list;
        block();
    }
}
```

- If value < 0
  abs(value) is the number of waiting processes
- `block()` suspends the process that invokes it
The `signal()` operation to eliminate busy waiting

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Wakeup(P) resumes the execution of process P

---

Deadlocks and Starvation: Implementation of semaphore with a waiting queue

- **PROCESS P0**
  - `wait(S);`
  - `wait(Q);`
  - `signal(S);`
  - `signal(Q);`

- **PROCESS P1**
  - `wait(Q);`
  - `wait(S);`
  - `signal(Q);`
  - `signal(S);`

**Say:** P0 executes `wait(S)` and then P1 executes `wait(Q)`

- P0 must wait till P1 executes `signal(Q)`
- P1 must wait till P0 executes `signal(S)`

Cannot be executed so deadlock
Semaphores and atomic operations

- Once a semaphore action has started
  - **No other process** can access the semaphore UNTIL
    - Operation has completed or process has blocked

- Atomic operations
  - Group of related operations
  - Performed without interruptions
    - Or not at all

---

Priority Inversion
Priority inversion

- Processes L, M, H (priority of L < M < H)
- Process H requires
  - Resource R being accessed by process L
  - Typically, H will wait for L to finish resource use
- M becomes runnable and preempts L
  - Process (M) with lower priority affects how long process H has to wait for L to release R

Priority inheritance protocol

- Process accessing resource needed by higher priority process
  - Inherits higher priority till it finishes resource use
  - Once done, process reverts to lower priority
CLASSIC PROBLEMS OF SYNCHRONIZATION

The bounded buffer problem

- Binary semaphore (mutex)
  - Provides mutual exclusion for accesses to buffer pool
  - Initialized to 1

- Counting semaphores
  - empty: Number of empty slots available to produce
    - Initialized to \( n \)
  - full: Number of filled slots available to consume
    - Initialized to 0
Some other things to bear in mind

- Producer and consumer must be ready before they attempt to enter critical section

- Producer readiness?
  - When a slot is available to add produced item
    - `wait(empty);` empty is initialized to `null`

- Consumer readiness?
  - When a producer has added new item to the buffer
    - `wait(full);` full initialized to `0`

The Producer

```c
do {
  produce item nextp
  wait till slot available
  wait(empty);
  wait(mutex);
  Only producer OR consumer can be in critical section
  add nextp to buffer
  signal(mutex);
  signal(full);
  Allow producer OR consumer to (re)enter critical section
  remainder section
} while (TRUE);
```

signal consumer that a slot is available
The Consumer

do {
    remove item from buffer
    (nextc)
    wait (full);
    wait (mutex);
    signal (mutex);
    signal (empty);
    consume nextc
} while (TRUE);

The Readers-Writers Problem
The Readers-Writers problem

- A database is **shared** among several concurrent processes

- Two types of processes
  - Readers
  - Writers

Readers-Writers: Potential for adverse effects

- If **two readers** access shared data simultaneously?
  - No problems

- If a **writer and some other reader** (or writer) access shared data simultaneously?
  - Chaos
Writers must have exclusive access to shared database while writing

- **FIRST readers-writers problem:**
  - No reader should wait for other readers to finish; simply because a writer is waiting
    - Writers may starve

- **SECOND readers-writers problem:**
  - If a writer is ready it performs its write ASAP
    - Readers may starve

Solution to the FIRST readers-writers problem

- **Variable** `int readcount`
  - Tracks how many readers are reading object

- **Semaphore** `mutex {1}`
  - Ensure mutual exclusion when `readcount` is accessed

- **Semaphore** `wrt {1}`
  1. Mutual exclusion for the writers
  2. First (last) reader that enters (exits) critical section
    - Not used by readers, when other readers are in their critical section
The Writer: When a writer signals either a waiting writer or the readers resume

\[
do \{ \\
\text{wait(wrt);} \\
\text{writing is performed} \\
\text{signal(wrt);} \\
\} \text{ while (TRUE);} \\
\]

When:
- writer in critical section and if \( n \) readers waiting
- 1 reader is queued on \( \text{wrt} \)
- \( (n-1) \) readers queued on \( \text{mutex} \)

The Reader process

\[
do \{ \\
\text{wait(mutex);} \\
\text{readcount++;} \\
\text{if (readcount ==1) \{} \\
\text{wait(wrt);} \\
\text{\}} \\
\text{signal(mutex);} \\
\text{reading is performed} \\
\text{wait(mutex);} \\
\text{readcount--;} \\
\text{if (readcount ==0) \{} \\
\text{signal(wrt);} \\
\text{\}} \\
\text{signal(mutex);} \\
\} \text{ while (TRUE);} \\
\]

When:
- writer in critical section and if \( n \) readers waiting
- 1 is queued on \( \text{wrt} \)
- \( (n-1) \) queued on \( \text{mutex} \)
The contents of this slide set are based on the following references
