Frequently asked questions from the previous class survey

- Scheduling
  - Is it common for processes to get through quanta and still not be completed?
- Critical sections
  - Why is this important: correctness, data corruption?
  - Can these cause starvation?
  - What happens if there is a context switch while process is in critical section?
  - If we know they are doing, can we execute them simultaneously?
  - Are hardware atomic instructions part of the ISA?
  - Libraries for critical section management?
  - Do they leverage ISA support?
  - Can critical section C-A be significantly larger in one process than another?

Topics covered in the lecture

- Synchronization hardware
- Using `TestAndSet` to satisfy critical section requirements
- Semaphores
- Classical process synchronization problems
- Midterm

Solving the critical section problem using locks

```c
    do {
        acquire lock
        critical section
        release lock
        remainder section
    } while (TRUE);
```

Possible assists for solving critical section problem

- Uniprocessor environment
  - Prevent interrupts from occurring when shared variable is being modified
    - No unexpected modifications!
- Multiprocessor environment
  - Disabling interrupts is time consuming
    - Message passed to ALL processors
Possible assists for solving critical section problem (2/2)

- Special atomic hardware instructions
  - Swap content of two words
  - Modify word

Swap()

```c
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Swap: Shared variable LOCK is initialized to false

```c
do {
    key = (key == TRUE) ? Swap(&lock, &key) : lock = FALSE;
} while (TRUE);
```

TestAndSet()

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

TestAndSet: Shared boolean variable lock initialized to false

```c
do {
    while (TestAndSet(&lock)) {};
} while (TRUE);
```

Entering and leaving critical regions using TestAndSet and Swap (Exchange)

```c
enter_region:
    TSL REGISTER, LOCK
    CMP REGISTER, #0
    JNE enter_region
    RET

leave_region:
    MOVE LOCK, #0
    RET
```

All Intel x86 CPUs have the XCHG instruction for low-level synchronization
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 \( i \)

- \( \text{waiting}[i] = \text{TRUE}; \)
- \( \text{key} = \text{TRUE}; \)
- while (\( \text{waiting}[i] \land \text{key} \)) {
  - \( \text{key} = \text{TestAndSet}(&\text{lock}); \)
- }
- \( \text{waiting}[i] = \text{FALSE}; \)

First process to execute TestAndSet will find key == false; EVERYONE else must wait!

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

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
  - if (\( j=i \)) {
    - \( \text{lock} = \text{FALSE}; \)
  } else {
    - \( \text{waiting}[j] = \text{FALSE}; \)
  }

Found a suitable waiting process

Mutual exclusion

- The variable waiting[i] can become false ONLY if another process leaves its critical section
- Only one waiting[i] is set to FALSE
Progress

- A process exiting the critical section
  1. Sets lock to FALSE
  2. \( \text{waiting}[j] \) to FALSE
- Allows a process that is waiting to proceed

Bounded waiting requirement

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

- \text{Scans} \ \text{waiting}[j] \text{ in the cyclic ordering} \ (i+1, i+2, \ldots, n, 0, \ldots, i-1)
- \text{ANY waiting process trying to enter critical section will do so in} (n-1) \text{ turns}

Semaphores

- Semaphore \( S \) is an integer variable
- Once initialized, accessed through \textbf{atomic} operations
  - \textbf{wait}()
  - \textbf{signal}()

Modifications to the integer value of semaphore execute indivisibly

\textbf{wait}(S) \{
  \textbf{while} \ (S<=0) \{
    ; //no operation
  \\
  S++;
\}
\}
\textbf{signal}(S) \{
  S=;
\}

Types of semaphores

- Binary semaphores
  - The value of \( S \) can be 0 or 1
    - Also known as \textbf{mutex locks}
- Counting semaphores
  - Value of \( S \) can range over an \textbf{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 S2 to execute only after S1 has executed

Semaphore `synch` is initialized to 0

```
wait(synch);
s1;
s2;
signal(synch);
```

PROCESS P1

PROCESS P2

The counting semaphore

- Controls access to a finite set of resource instances
- **INITIALIZED** to the number of resources available
- Resource Usage
  - `wait()`: To use a resource
  - `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

- **{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

Defining the semaphore

```c
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();
    }
}
```

`block()` suspends the process that invokes it

If value < 0, `abs(value)` is the number of waiting processes.

The `signal()` operation to eliminate busy waiting

```c
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
```c
wait(S);
wait(Q);
signal(S);
signal(Q);
```

PROCESS P1
```c
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)` to resume execution.
P1 must wait till P0 executes `signal(S)` to resume execution.

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

- 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

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
    - Initialized to N
  - full: Number of filled slots available
    - 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 \( N \)
- 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);
    add nextp to buffer
    signal(mutex);
    signal(full);
    remainder section
} while (TRUE);
```

The Consumer
```c
do {
    wait(till slot available for consumption)
    wait(mutex);
    remove item from buffer (nextc)
    consume nextc
    signal(mutex);
    signal(empty);
    signal producer that a slot is available to add
} while (TRUE);
```
The contents of this slide set are based on the following references: