Frequently asked questions from the previous class survey

- Thread TCB: Must each thread have one? Where does it reside? Is there a creation overhead? What happens to the stack when a thread is done executing?
- Thread Models
  - Many-to-many: How does the kernel multiplex?
- Threads: sleep() vs. wait()
- Relationship with execution on cores: Processes and Threads
Synchronization: What we will look at

- Synchronization primitives
- Classical Synchronization problems
- Why?
- Race Conditions
- Critical Sections
- Hardware assists
- Critical Section problem & solution requirements

Topics covered in the lecture

- Critical section
- Critical section problem
- Peterson’s solution
- Hardware assists
Reasoning about interleaved access to shared state: Too much milk!

<table>
<thead>
<tr>
<th>Roommate 1’s actions</th>
<th>Roommate 2’s actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00 Look in fridge; out of milk</td>
<td>Look in fridge; out of milk</td>
</tr>
<tr>
<td>3:05 Leave for store</td>
<td>Leave for store</td>
</tr>
<tr>
<td>3:10 Arrive at store</td>
<td>Arrive at store</td>
</tr>
<tr>
<td>3:15 Buy milk</td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:20 Arrive home; put milk away</td>
<td>Arrive home; put milk away</td>
</tr>
<tr>
<td>3:25</td>
<td></td>
</tr>
<tr>
<td>3:30</td>
<td></td>
</tr>
</tbody>
</table>

Oh no!

It is not enough to be industrious. So are the ants. The question is: What are we industrious about?

—Henry David Thoreau

Process Synchronization
Process synchronization

- How can processes pass information to one another?
- Make sure two or more processes do not get in each other’s way
  - E.g., 2 processes in an airline reservation system, each trying to grab the last seat for a different passenger
- Ensure proper sequencing when dependencies are present

Applicability to threads

- Passing information between threads is easy
  - They share the same address space of the parent process
- Other two aspects of process synchronization are applicable to threads
  - Keeping out of each other’s hair
  - Proper sequencing
A look at the producer consumer problem

while (true) {
    while (counter == BUFFER_SIZE) {
        /*do nothing */
    }
    buffer[in] = nextProduced
    in = (in +1)%BUFFER_SIZE;
    counter++;
}

while (true) {
    while (counter == 0) {
        /*do nothing */
    }
    nextConsumed = buffer[out]
    out = (out +1)% BUFFER_SIZE;
    counter--;
}

Implementation of ++/-- in machine language

```
counter++
    register1 = counter
    register1 = register1 + 1
    counter   = register1
```

```
counter--
    register2 = counter
    register2 = register2 - 1
    counter   = register2
```
Lower-level statements may be interleaved in any order

Producer execute: register1 = counter
Producer execute: register1 = register1 + 1
Producer execute: counter = register1

Consumer execute: register2 = counter
Consumer execute: register2 = register2 - 1
Consumer execute: counter = register2

The order of statements within each high-level statement is preserved
Lower-level statements may be interleaved in any order (counter = 5)

Producer execute: register1 = counter
{register1 = 5}

Producer execute: register1 = register1 + 1
{register1 = 6}

Consumer execute: register2 = counter
{register2 = 5}

Consumer execute: register2 = register2 - 1
{register2 = 4}

Producer execute: counter = register1
{counter = 6}

Consumer execute: counter = register2
{counter = 4}

Counter has incorrect state of 4

Lower-level statements may be interleaved in any order (counter = 5)

Producer execute: register1 = counter
{register1 = 5}

Producer execute: register1 = register1 + 1
{register1 = 6}

Consumer execute: register2 = counter
{register2 = 5}

Consumer execute: register2 = register2 - 1
{register2 = 4}

Consumer execute: counter = register2
{counter = 4}

Producer execute: counter = register1
{counter = 6}

Counter has incorrect state of 6
Race conditions

- Several processes access and manipulate data concurrently
- **Outcome** of execution *depends* on
  - Particular **order** in which accesses take place
- Debugging programs with race conditions?
  - Painful!
  - Program runs fine most of the time, but once in a rare while something weird and unexpected happens
Race condition: Example

- When process wants to print file, adds file to a special **spooler directory**
- Printer daemon periodically checks to see if there are files to be printed
  - If there are, print them
- In our example, spooler directory has a large number of slots
- Two variables
  - `in`: Next free slot in directory
  - `out`: Next file to be printed

Race condition: Example

- In jurisdictions where Murphy’s Law hold ...
- Process A reads `in`, and stores the value 7, in local variable `next_free_slot`
- Context switch occurs
- Process B also reads `in`, and stores the value 7, in local variable `next_free_slot`
  - Stores name of the file in slot 7
- Process A context switches again, and stores the name of the file it wants to print in slot 7
Race condition: Example

- Spooler directory is internally consistent

- But process B will never receive any output
  - User B loiters around printer room for years, wistfully hoping for an output that will never come ...

The kernel is subject to several possible race conditions

- E.g.: Kernel maintains list of all open files
  - 2 processes open files simultaneously
  - Separate updates to kernel list may result in a race condition

- Other kernel data structures
  - Memory allocation
  - Process lists
  - Interrupt handling
Critical Section

- **Concurrent accesses** to shared resources can lead to unexpected or erroneous behavior
- **Parts of the program** where the shared resource is accessed thus need to be protected
  - This protected section is the **critical section**
Critical-Section

- System of \( n \) processes \( \{P_0, P_1, \ldots, P_{n-1}\} \)
- Each process has a segment of code (critical section) where it:
  - Changes common variables, updates a table, etc
- No two processes can execute in their critical sections at the same time

The Critical-Section problem

- Design a protocol that processes can use to cooperate
- Each process must request permission to enter its critical section
  - The entry section
General structure of a participating process

\begin{align*}
do & \{ \\
& \text{entry section} \\
& \text{critical section} \\
& \text{exit section} \\
& \text{remainder section} \\
\} \text{ while (TRUE);}
\end{align*}

Requirements for a solution to the critical section problem
Requirements for a solution to the critical section problem

1. Mutual exclusion
2. Progress
3. Bounded wait

PROCESS SPEED
- Each process operates at non-zero speed
- Make no assumption about the relative speed of the \( n \) processes

Mutual Exclusion

- Only one process can execute in its critical section
- When a process executes in its critical section
  - No other process is allowed to execute in its critical section
Mutual Exclusion: Depiction

- Process A enters critical section at T1
- Process B attempts to enter critical section at T2
  - B blocked by Process A
- A exits critical section at T3
- B enters critical section at T3
  - B exits critical section at T4

Progress

- **{C1} If No** process is executing in its critical section, and ...
- **{C2} Some** processes wish to enter their critical sections

- **Decision** on who gets to enter the critical section
  - Is made by processes that are **NOT** executing in their remainder section
  - Selection **cannot be postponed indefinitely**
Bounded waiting

- **After** a process has made a **request** to enter its critical section
  - AND **before** this request is granted

- **Limit number** of times other processes are allowed to enter their critical sections

Approaches to handling critical sections in the OS

- **Nonpreemptive kernel**
  - If a process runs in kernel mode: no preemption
  - **Free** from race conditions on kernel data structures

- **Preemptive kernels**
  - Must ensure shared kernel data is free from race conditions
  - **Difficult** on SMP (Symmetric Multi Processor) architectures
    - 2 processes may run simultaneously on different processors
Kernels: Why preempt?

- Suitable for real-time
  - A real-time process may preempt a kernel process

- More responsive
  - *Less risk* that kernel mode process will run arbitrarily long

**Peterson’s Solution**

Software based solution
Peterson’s Solution

- **Software solution** to the critical section problem
  - Restricted to two processes

- No guarantees on modern architectures
  - Machine language instructions such as load and store implemented differently

- Good algorithmic description
  - Shows how to address the 3 requirements

---

*Peterson’s Solution: The components*

- Restricted to two processes
  - \( P_i \) and \( P_j \) where \( j = 1 - i \)

- **Share** two data items
  - \( \text{int turn} \)
    - Indicates whose turn it is to enter the critical section
  - \( \text{boolean flag}[2] \)
    - Whether process is ready to enter the critical section
Peterson’s solution: Structure of process \( P_i \)

```c
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn==j) {};
    critical section
    flag[i] = FALSE;
} while (TRUE);
```

Peterson’s solution: Mutual exclusion

- \( P_i \) enters critical section only if
  - flag[j] == false OR turn == i

- If both processes execute in critical section at the same time
  - flag[0] == flag[1] == true
  - But turn can be 0 or 1, not BOTH

- If \( P_j \) entered critical section
  - flag[j] == true AND turn == j
  - Will persist as long as \( P_j \) is in the critical section
Peterson’s Solution:
Progress and Bounded wait

- $P_i$ can be stuck only if $\text{flag}[j] == \text{true}$ AND $\text{turn} == j$
  - If $P_j$ is not ready: $\text{flag}[j] == \text{false}$, and $P_i$ can enter
  - Once $P_j$ exits: it resets $\text{flag}[j]$ to false

- If $P_j$ resets $\text{flag}[j]$ to true
  - Must set $\text{turn} = i$

- $P_i$ will enter critical section (progress) after at most one entry by $P_j$ (bounded wait)
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

\[
\text{do } \{ \\
\text{key = TRUE; } \\
\text{while (key == TRUE) } \{ \\
\text{ Swap(&lock, &key) } \\
\text{\}} \\
\text{critical section } \\
\text{lock = FALSE; } \\
\text{remainder section } \\
\text{key = TRUE; } \\
\text{while (key == TRUE) } \{ \\
\text{ Swap(&lock, &key) } \\
\text{\}} \\
\text{lock is a SHARED variable } \\
\text{key is a LOCAL variable } \\
\text{if two Swap() are executed } \\
\text{simultaneously, they will be executed } \\
\text{sequentially in some arbitrary order} \\
\}\text{ while (TRUE);} \\
\]

TestAndSet()

\[
\text{boolean TestAndSet(boolean *target ) } \{ \\
\text{ boolean rv = *target; } \\
\text{ *target = TRUE; } \\
\text{ return rv;} \\
\} \\
\]

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

\[
do \{
\text{while (TestAndSet(&lock)) \{};}
\]

\[
\text{critical section}
\]
\[
\text{lock} = \text{FALSE};
\]

\[
\text{remainder section}
\]
\[
\} \text{ while (TRUE);};
\]

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

---

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

**enter** _region:_

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

```
enter_region:
MOVE REGISTER, #1
XCHNG REGISTER,LOCK
CMP REGISTER, #0
JNE enter_region
RET
```

**leave** _region:_

```
MOVE LOCK, #0
RET
```

```
leave_region:
MOVE LOCK, #0
RET
```

All Intel x86 CPUs have the XCHG instruction for low-level synchronization

---

**September 18, 2018**

**Professor: SHRIDEEP PALLICKARA**

**CS370: Operating Systems [Fall 2018]**

**Dept. Of Computer Science, Colorado State University**
The contents of this slide set are based on the following references


