Synchronization

Weeks 4 - 6
Announcements & Comments

• Read the whole Synchronization Chapter in the book; it contains important material for the second assignment

• My slides go beyond the chapter on synchronization, read them carefully, **especially with regards to the second assignment (read forward, i.e., beyond the material covered in the lecture)!!**

• I may not cover the slides in the exact sequence presented
Announcements & Comments

• About the assignment’s locks see these slides
• About the assignment’s condition variables see these slides and read monitor section in book
• For pet synchronization read classical synchronization section in the book
• Pet synchronization solution et al. should be generic
• You can execute your synchronization mechanisms and your solution to the pet problem via the kernel’s boot menu (see handout)

• Design a generic solution to the pet problem (we will test for that)

• We will compare all submitted assignments to identify cheaters …

• Do NOT change the boot menu or the test code that it invokes (we plan to add additional test code, beyond the one in your distribution)
• Only fill in the blanks (function stubs) in the respective files (do not worry about how these functions are invoked)
• Start Assignment 1 from a new and clean distribution (make sure you do not unpack it over your old distribution, since certain files may or may not be overwritten)
  – Start from scratch with a new distribution
  – If you wish, keep your debugging statements
  – You will build on the synchronization primitives you develop
• You may have to modify all kinds of OS/161 code, not just the pieces we explicitly point you to
• Localized understanding is essential; understanding the whole kernel is not required to solve the assignment
Before We Start

• OS/161 processes are single threaded
• OS/161 processes are realized via the “threads structure” (see earlier slides), but are NOT threads in the sense introduced in my lecture

• Nothing prevents you from making them multi-threaded – we wont ask you to do that
Motivation

- Processes may want to **pass on information**, e.g., UNIX pipe “ls -l | grep *.c”.
- Process A may **require to wait for output** of process B, e.g., printer spooler waits for files to print.
- **Coordinate critical activities** e.g., memory allocation.
- **Share and access** data elements
- Keep track of the **number of times an activity is execution**, e.g., the number of writing transactions in a DBMS
Bounded Buffer

Examples

- Printer queues
- Device buffers
- Shared buffers or queues to pass information between processes

The following discussion applies equally to process and to threads.
//shared data
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
//initial values
int in = 0;
int out = 0;
int counter = 0;

Example:

out = 1
in = 5
counter = 4
Bounded Buffer: Producer

item \textit{nextProduced};
while (TRUE) {
    while (counter == BUFFER\_SIZE) ; /* FULL - do nothing */
    buffer[in] = \textit{nextProduced};
    in = (in + 1) \% BUFFER\_SIZE;
    counter++;
}

Bounded Buffer: Consumer

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

ECE 344 Operating Systems
Machine-level Implementation

• Implementation of “counter++”
  \[ \text{register}_1 = \text{counter} \]
  \[ \text{register}_1 = \text{register}_1 + 1 \]
  \[ \text{counter} = \text{register}_1 \]

• Implementation of “counter—”
  \[ \text{register}_2 = \text{counter} \]
  \[ \text{register}_2 = \text{register}_2 - 1 \]
  \[ \text{counter} = \text{register}_2 \]
Bounded Buffer

• If both the producer and consumer attempt to update the buffer *concurrently*, the assembly language statements may get interleaved.

• Interleaving *depends* upon how the producer and consumer are *scheduled*. 

```plaintext
producer  consumer
```

ECE 344 Operating Systems 13
Possible Execution Patterns

Producer

Context Switch

Consumer

Context Switch

Context Switch

Context Switch

Producer

Consumer
Interleaved Execution

- Assume `counter` is 5 and one interleaved execution of producer and consumer code (i.e., `counter++` and `counter--`):

  - **P:** \( r1 = \text{counter} \) \( (\text{register}_1 = 5) \)
  - **P:** \( r1 = r1 + 1 \) \( (\text{register}_1 = 6) \)
  - **C:** \( r2 = \text{counter} \) \( (\text{register}_2 = 5) \)
  - **C:** \( r2 = r2 - 1 \) \( (\text{register}_2 = 4) \)
  - **P:** \( \text{counter} = r1 \) \( (\text{counter} = 6) \)
  - **C:** \( \text{counter} = r2 \) \( (\text{counter} = 4) \)

- The value of `counter` may be either 4 or 6, where the correct result should be 5.
Race Condition

• **Race condition**: The situation where several processes or threads access and manipulate shared data concurrently, while the final value of the shared data depends upon which process finishes last.

• In our example for P last, result would be 6, and for C last, result would be 4.

• To prevent race conditions, concurrent processes must be synchronized.
The Moral of this Story

• The statements
  counter++; counter--;
must be performed *atomically*.

• Atomic operation means an operation that
  completes in its entirety without interruption.

• This is achieved through *synchronization
  primitives* (semaphores, locks, condition
  variables, monitors …).
Synchronization: Overview

- More formal definition of problem (the critical section problem)
- Simple solutions to this problem
  - Software solutions to this problem (defer till later)
  - Hardware support for synchronization (defer till later)
- Locking, semaphores, condition variables
- Higher-level synchronization primitives
- Common synchronization problems
The Critical-Section Problem

• **n** processes all competing to **use some shared data**.

• Each process has a code segment, called **critical section**, in which the shared data is accessed.

• **Problem**: ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

• Sometimes also called **critical region** – don’t confuse this with our book’s critical region **construct**.
A Simple Solution: Disabling of Interrupts

• Context switches come about through interrupts (e.g., clock or other interrupts)
• So how about disabling interrupts while `counter++` is executed?
  – Should user really be allowed to do that?
  – What does that mean in a multi-CPU context?
• Inside kernel code this may be acceptable
• This is the mechanism employed by OS/161 to achieve atomicity in the kernel – for short pieces of code
• Your mission – should you accept it 😊 - is to implement higher-level synchronization mechanisms in OS/161 (locks and condition variables)
Meta Comment

• We will skip a number of sections in the book at this point and come back to them later
• We are skipping software-based solutions to the critical section problem for now (read them)
• We are skipping hardware features in support of critical section
• These solutions are based on mechanisms that require busy waiting
Semaphores

• Higher-level synchronization mechanism
• Higher than disabling interrupts
  – Fine for short sequences of code in kernel
  – Not fine for application-level use
Semaphores

- **Semaphore S**, integer variable
- can only be accessed via two *indivisible* (atomic) operations

1. *wait* (S): // historically a.k.a. P(S)
   ```
   while
   do nothing;
   ```
   atomic for S > 0

2. *signal* (S): // historically a.k.a. V(S)
   ```
   ```
   e.g., by disabling interrupts
   busy waiting loop
wait(S)

EI – enable interrupt
DI – disable interrupt

loop:

DI

if S <= 0 then {
    EI;
    goto loop
}
else {
    S—;
    EI;
}

critical section for S <= 0
critical section for S > 0
Critical Section of $n$ Processes

- **Shared data:**
  
  semaphore mutex; // initially mutex = 1

- **Process $P_i$:**

  
  do {
  wait(mutex);
  critical section
  signal(mutex);
  remainder section
  } while (TRUE);

  
  Timeline:

  P$_1$: m=1
  wait(m)
  m -- //m=0
  wait(m)
  P$_2$: m=1
  m <= 0 ?
  m -- //m=0
  time
  signal(m)
  m ++ //m=1
semaphore m=1

P_1: m=1

\[ m \leq 0 ? \]
wait(m)

P_2: m=1

\[ m \leq 0 ? \]
wait(m)

P_3: m=1

\[ m \leq 0 ? \]
m-- //m=0

P_4: m=1

\[ m \leq 0 ? \]
wait(m)

wait (S):

while

\[ S \leq 0 \]
do nothing;

m++ //m=1
semaphore m=3

P₁: m=3
2
wait(m)
m ≤ 0 ?
m-- //m=1

P₂: m=3
1
wait(m)
m ≤ 0 ?
m-- //m=2

P₃: m=3
3
wait(m)
m ≤ 0 ?
m-- //m=0

P₄: m=3

wait(S):

while

donthing;

ECE 344 Operating Systems
semaphore m=0

P1: m=0
   wait(m)
   m ≤ 0 ?

P2: m=0
   wait(m)
   m ≤ 0 ?

P3: m=0
   wait(m)
   m ≤ 0 ?

P4: m=0
   wait(m)
   m ≤ 0 ?

wait (S):

Rien ne va plus!

while
do nothing;
Semaphore Implementation

• Variant that avoids busy waiting
• Define a semaphore as a record (shared data)

```c
typedef struct {
    int value;
    struct process *L;
} semaphore;
```

• Assume two simple operations:
  – `block()` suspends the process that invokes it.
  – `wakeup(P)` resumes the execution of a blocked process $P$. 
Implementation Alternative

\textbf{wait}(S):

\begin{align*}
\text{atomic} & \quad \text{S.value--} \\
& \quad \text{if} (\text{S.value} < 0) \quad \text{add this process to S.L} \\
& \quad \text{block();} \\
\end{align*}

\textbf{signal}(S):

\begin{align*}
\text{atomic} & \quad \text{S.value++} \\
& \quad \text{if} (\text{S.value} \leq 0) \quad \text{remove a process P from S.L} \\
& \quad \text{wakeup(P);} \\
\end{align*}
Example (expressed as timeline)

Semaphore m;
m=1; // shared data

<table>
<thead>
<tr>
<th>P_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait(m)</td>
</tr>
<tr>
<td>m.v-- // m.v= -1</td>
</tr>
<tr>
<td>m.v &lt; 0 ?</td>
</tr>
<tr>
<td>enqueue + block</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P_j</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait(m)</td>
</tr>
<tr>
<td>m.v-- // m.v= 0</td>
</tr>
<tr>
<td>m.v &lt; 0 ?</td>
</tr>
<tr>
<td>dequeue + wakeup</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P_k</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait(m)</td>
</tr>
<tr>
<td>m.v-- // m.v= -2</td>
</tr>
<tr>
<td>m.v &lt; 0 ?</td>
</tr>
<tr>
<td>enqueue + block</td>
</tr>
</tbody>
</table>

Semaphore m;
do {
   \% P_i
   wait(m);
   critical section
   signal(m);
   remainder section
} while (TRUE);
Semaphores in OS/161

- Defined in `src/kern/thread/synch.c` and `src/kern/include/synch.h`
- Based on Dijkstra semantic with P/V (\textit{proberen} (try) / \textit{verhogen} (increase)) operations instead of wait/signal
Semaphore Implementation in OS/161

```c
void P(struct semaphore *sem)
{
    int spl;
    assert(sem != NULL);
    spl = splhigh();
    while (sem->count == 0) {
        thread_sleep(sem);
    }
    assert(sem->count > 0);
    sem->count--;
    splx(spl);
}
```

*Is like our wait(sem).*
Semaphore Implementation in OS/161

```c
void V(struct semaphore *sem)
{
    int spl;
    assert(sem != NULL);
    spl = splhigh();
    sem->count++; 
    assert(sem->count > 0);
    thread_wakeup(sem);
    splx(spl);
}
```

Wakes up all threads waiting on `sem`

Is like our `signal(sem)`.
Semaphore as Synchronization Tool

- Execute $B$ in $P_j$ only after $A$ executed in $P_i$
- Use semaphore **flag initialized to 0**
- Code:

  $P_i$ \hspace{1cm} $P_j$
  $A$ \hspace{1cm} \texttt{wait(flag)}
  \texttt{signal(flag)} \hspace{1cm} $B$
Careful: Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

- Let $S$ and $Q$ be two semaphores initialized to 1

  ```
  P_0
  wait(S);
  wait(Q);
  M
  signal(S);
  signal(Q)

  P_1
  wait(Q);
  wait(S);
  M
  signal(Q);
  signal(S)
  ```

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
Bounded Buffer with Semaphores

- Shared data:
  
  ```plaintext
  semaphore mutex = 1;  // exclusive access
  semaphore empty = N;  // number of empty slots
  semaphore full = 0;    // number of full slots
  ```

- Semaphores initialized to 1 and used to serialize access to a critical section are sometimes called **binary semaphores** ≠ **locks**
Bounded Buffer: Producer

item \textit{nextProduced};
while (TRUE) {
    wait (empty);
    wait (mutex);
    insert(\textit{nextProduced});
    signal (mutex);
    signal (full);
}

item \textit{nextConsumed};
while (TRUE) {
    wait (full);
    wait (mutex);
    \textit{nextConsumed} = remove();
    signal (mutex);
    signal (empty);
}

- Buffer implemented as a linked list
Bounded Buffer: Producer (broken)

item nextProduced;
while (TRUE) {
    while (counter == BUFFER_SIZE) {
        /* FULL - do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
    }
}
Semaphores & Locks
Semaphores, Locks & Mutexes

- **Counting** semaphores vs. **binary** semaphores
  - semaphore integer takes on various values
  - semaphore integer takes on values 0 and 1
  - *Can a counting semaphore be implemented based on a binary semaphore?*

- **A binary semaphore is not a lock**
  - But maybe used just like a lock (other use patterns also possible)
Semaphores, Locks & Mutexes

• **Mutex** often refers to a *locking mechanism* available in user-space (user-level threads)
  – Various different kinds …

• Term **lock** is used to also refer to a *locking mechanism* (remember presentation of semaphores in class – lock/unlock – wait/signal)
  – However, wait/signal can be used by two different processes
  – Lock/corresponding unlock **must be called from same process**
Mutex – Mutual Exclusion
(What the assignment calls a lock.)

• A semaphore that allows only one process inside the critical section is often called a mutex
• Semaphores’ ability to count not required in the application semantic
• Mutexes are used exclusively to manage mutual exclusion of critical section (i.e., lock and unlock)
• Easy and efficient to implement (therefore attractive for user-level thread packages)
• Mutex knows one of two states, 0 or 1 – unlocked, locked
• If TSE instruction available, mutexes can be easily implemented in user space (discussed later)
Semaphores in OS/161

• For implementing locks/CVs it maybe helpful to study the semaphore implementation in OS/161

• Defined in `src/kern/thread/synch.c` and `src/kern/include/synch.h`

• Based on Dijkstra semantic with P/V (`proberen` (try) / `verhogen` (increase)) operations instead of wait/signal
Desirable & Undesirable Properties of Lock Implementations

• Improper use of locks
  – Locking a non-initialised mutex (lock)
  – Locking a mutex that you already own
  – Unlocking a mutex that you don’t own

• As always in this context, it’s the user’s responsibility to prevent this from happening

• Some thread implementations do check for these conditions and signal the problem

• Note, that for semaphore (binary semaphores) the above properties are not meant to be enforced
Mutexes/Locks in OS/161

```c
struct lock{
    char * name;
    struct thread *holder;
};

struct lock *
    lock_create (const char *name);

void lock_acquire (struct lock *);

void lock_release (struct lock *);

int lock_do_i_hold (struct lock *);

void lock_release (struct lock *);
```
Towards Higher-level Synchronization Constructs

• Getting the wait/signals correct is not easy
• Higher-level languages help programmer synchronize the applications, e.g.,
  – Java’s synchronize (single threaded access of methods of class guaranteed)
  – 1975 introduction of monitor by Hoar et al.
  – See also “critical region construct” in our text book
Monitor

- **High-level synchronization** construct that allows the **safe sharing** of an abstract data type among concurrent processes.

```plaintext
monitor monitor-name {
    shared variable declarations
    procedure body P1 (...) { . . . }
    ...
    procedure body P2 (...) { . . . }
    procedure body Pn (...) { . . . }

    { initialization code }
}
```

- Access to monitor code is **mutually exclusive** for caller
Schematic View of a Monitor

entry queue

shared data

operations

initialization code
Condition Variables

• To allow a process to wait within the monitor, a **condition variable** must be declared, as `condition x, y;`

• Condition variables can only be used with the operations **wait** and **signal**.
  
  – `x.wait()` means that the process invoking this operation is suspended until another process invokes `x.signal();`
  
  – `x.signal` resumes **exactly one** suspended process. If no process is waiting, then the **signal** operation **has no effect** (**unlike a semaphore’s signal(...).**)
Monitor With Condition Variables

- Entry queue
- Shared data
- Queues associated with x, y conditions
- Operations
- Initialization code
Dining-Philosophers Problem
Dining Philosophers Example

```c
monitor dp {
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i)
    void putdown(int i)
    void test(int i)
    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}
```
Dining Philosophers

void pickup(int i) {
    state[i] = hungry;
    test[i];
    if (state[i] != eating)
        self[i].wait();
}

void putdown(int i) {
    state[i] = thinking;
    test((i+4) % 5);
    test((i+1) % 5);
}

\[ 
\begin{array}{c}
\text{i} \\
0 & 1 & 2 \\
4 & 3 \\
\end{array} 
\]
Dining Philosophers

```c
void test(int i) {
    if ((state[(i + 4) % 5] != eating) &&
        (state[i] == hungry) &&
        (state[(i + 1) % 5] != eating)) {
        state[i] = eating;
        self[i].signal();
    }
}
```
Example: Putdown

```c
void putdown(int i) {
    state[i] = thinking;
    test((i+4) % 5);
    test((i+1) % 5);
}
```

(2): putdown(i)

test(A) …

test(A):

If left ng. not eating and “A” is hungry and right ng. not eating then set “A” to eat

signal(A) // wakeup A

(1): pickup(A)

- A tried picking up a ch. stick
- Failed and put itself to sleep
NB: OS/161 CVs

• The notion of CVs in the context of monitors correspond to the notion of CVs asked from you in this current assignment (i.e., in OS/161)

• The difference between a monitor CV and an OS/161 CV is
  – For a **monitor the lock** that protects the monitor data structure (i.e., realizes mutual exclusion) **is implicit** – by virtue of the construct being a monitor
  – For the OS/161 / second assignment CV the lock is **explicit** and is passed as argument to the CV API / function calls you have to implement
Condition Variables

• Monitor’s signal & wait are condition variables
• CVs also exist outside monitors, e.g., in Pthreads and in OS/161, at least, hopefully soon …
• CVs are a way for threads to notify each other (a notification system for threads)
• Instead of CVs threads could poll variables (i.e., lock, query, unlock, which is not efficient)
• Read the specification in synch.h, which tells you how to implement CVs
CV Example

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 3
#define TCOUNT 10
#define COUNT_THRES 12

int count = 0;
int thread_ids[3] = {0,1,2};

pthread_mutex_t count_lock=
    PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t count_hit_threshold=
    PTHREAD_COND_INITIALIZER;
```

- This is an example based on pthreads.
- Here the “monitor” lock is made explicit
- This is not a monitor
- This is very similar to the OS/161 API of CVs
CV Example

main(void) {
    int      i;
    pthread_t threads[3];

    pthread_create(&threads[0], NULL, inc_count, (void *)&thread_ids[0]);
    pthread_create(&threads[1], NULL, inc_count, (void *)&thread_ids[1]);
    pthread_create(&threads[2], NULL, watch_count, (void *)&thread_ids[2]);

    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    return 0;
}
CV Example

void *inc_count(void *idp) {
    int i=0, save_state, save_type;
    int *my_id = idp;
    for (i=0; i<TCOUNT; i++) {
        pthread_mutex_lock(&count_lock);
        count++;
        printf(" … ");
        if (count == COUNT_THRES) {
            printf(" … ");
            pthread_cond_signal(&count_hit_threshold);
        } // ends if
        pthread_mutex_unlock(&count_lock);
    } // ends for
    printf(" … ");
    if (count == COUNT_THRES) {
        printf(" … ");
        pthread_cond_signal(&count_hit_threshold);
    } // ends if
    pthread_mutex_unlock(&count_lock);
    return(NULL); } // ends inc_count procedure

However, if predictable scheduling behavior is required, then that mutex should be locked by the thread calling pthread_cond_signal().
void *watch_count(void *idp) {
    int i=0, save_state, save_type;
    int *my_id = idp;
    printf("watch_count(): thread %d\n", *my_id);
    pthread_mutex_lock(&count_lock);
    while (count < COUNT_THRES) {
        pthread_cond_wait(&count_hit_threshold, &count_lock);
        printf(" … ");
    }
    pthread_mutex_unlock(&count_lock);
    return(NULL);  // ends watch_count  }

CV Example
Synchronization

Recap on Semaphores/Locks and CVS
Synchronization Mechanisms: Overview

• Semaphores (binary, counting)
  – Enforce **mutually exclusive** use of resources
  – Enforce **arbitrary execution patterns** (e.g., **sequential** or **ordering** constraints)
  – Enforce **synchronization constraints** (e.g., full, empty, ..)

• Locks and mutexes
  – Enforce **mutually exclusive** use of resources, **exclusively**

• Condition variables
  – Enforce **waiting for events and conditions** (e.g., value of data)

• Monitors (& critical region construct)
  – **Higher-level** synchronization primitives
  – **Condition variables** introduced in this context
Common Use-patterns of the Above

```plaintext
wait(mutex);
    ... critical section
signal(mutex);

wait (empty);
wait (mutex);
    insert(...);
signal (mutex);
signal (full);

A  signal(flag)
   wait(flag)   B

lock(l)
    ... critical section
unlock(l);
```
Classical Problems of Synchronization

• (Bounded-Buffer Problem)
  – Already covered based on semaphores
• Readers and Writers Problem
• Dining-Philosophers Problem
Readers-Writers Problem

• The problem
  – Many readers may access critical section concurrently
  – Writer requires exclusive access to critical section
    • \textit{If readers are in CS and a writer comes along, CS is drained}
    • If readers are in CS and a writer comes along, writer waits until there are no further readers

• Shared data
  \textbf{semaphore} \texttt{mutex, wrt};

• Initially
  \texttt{mutex = 1, wrt = 1;}
  \texttt{int readcount = 0;}
Readers-Writers Problem **Writer Process**

- Exclusive access to critical section must be enforced via the semaphore, `wrt`

```c
wait(wrt);  //write lock
...
writing is performed
...
signal(wrt);
```
**Readers-Writers Problem Reader Process**

```c
wait(mutex);
readcount++;
if (readcount == 1)
  wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount--;
if (readcount == 0)
  signal(wrt);
signal(mutex);
```

- Concurrent access by other readers
- Counts the number of readers in CS

**For 1st reader:**
- If CS is not locked, enter and read
- …otherwise, wait on writer exiting, i.e., lock writer lock (wrt)

**For last reader exiting CS:**
- unlock writer lock
Dining-Philosophers Problem

- Shared data
  
  ```
  semaphore chopstick[5];
  ```
- Initially all values are 1
Dining-Philosophers Problem

do { // Philosopher i
    wait(chopstick[i])
    wait(chopstick[(i+1) % 5])
    ...
    eat
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
    think
    ...
} while (TRUE);
Example Execution

wait(1)  wait(2)
wait(2)  wait(3)

wait(5)  wait(1)
Example Execution: Problem Case

1
   wait(1)
   wait(5) ⋮ → wait(2) 2
signal(5) ⋮ signal(2)
signal(1) wait(4) ⋮ signal(3)
   wait(3)
Hardware-based Solutions for Synchronization
Atomicity
Semaphore Implementation in OS/161

```c
void P(struct semaphore *sem)
{
    int spl;
    assert(sem != NULL);
    spl = splhigh();
    while (sem->count==0) {
        thread_sleep(sem);
    }
    assert(sem->count>0);
    sem->count--;
    splx(spl);
}
```

Puts thread to sleep and ...

Why is there a while loop?

Is like our wait(sem).
Semaphores

• **Semaphore S**, integer variable
• can only be accessed via two *indivisible* (atomic) operations

1. *wait* (S):  // historically a.k.a. P(S)

   ```
   while S <= 0 do
   do nothing;
   ```

2. *signal* (S):  // historically a.k.a. V(S)

   ```
   S++;
   ```

    | atomic
    | e.g., by disabling interrupts

Busy waiting loop
Implementation Alternative

\textit{wait}(S):

\begin{itemize}
  \item \texttt{S.value--;}
  \item \texttt{if (S.value < 0) \{ add this process to S.L; block(); \}}
\end{itemize}

\textit{signal}(S):

\begin{itemize}
  \item \texttt{S.value++;}
  \item \texttt{if (S.value \leq 0) \{ remove a process P from S.L; wakeup(P); \}}
\end{itemize}
Our Atomicity Assumption in Semaphores

• Our assumption was not obvious and not fully true
• Now, with a little help from the hardware
  – TSE RX, Lock  // Atomic test-and-set
  – Read Lock into register, RX
  – Store a non-zero value into memory location Lock
  – i.e., no other process can access memory location until the operation has completed
  – CPU executing TSE, locks the memory bus to prevent access of memory from other CPUs (if multi CPU sys.)
• Supported by many hardware platforms (not by MIPS-1, ☹; but there we have splhigh/splx)
Synchronization Hardware

- TSE modifies the content of a word atomically.
- As pseudo code below.
- Implemented by one hardware instruction, **TSE**.

```c
Boolean TestAndSet(Boolean &target) {
    Boolean rv = target;
    target = true;
    return rv;
}
```

Atomicity **guaranteed** by hardware!
User-level Implementation

**Lock:**  
TSE R, MUTEX  
CMP R, #0  
JZE ok  
CALL thread_yield  
JMP Lock  
ok RET

**UN_Lock:**  
MV MUTEX,#0  
RET

Applies to threads discussion only.
Mutual Exclusion with Test-and-Set

- **Shared data:**
  
  Boolean lock = false;

- **Process** $P_i$

  do {
  
  while (TestAndSet(lock)) ;
  
  critical section
  
  lock = false;
  
  remainder section
  
  }

  Busy waiting

  enter_section:
  
  TSE R, Lock
  
  CMP R, #0
  
  JNE enter_section
  
  RET

  leave_section:
  
  MOVE Lock, #0
  
  RET
Example: Timeline

Shared data: \texttt{lock = false;}

\begin{itemize}
  \item \texttt{P_1:}
    \begin{itemize}
    \item \texttt{while(TestAndSet(Lock))}
    \item \texttt{busy waiting}
    \end{itemize}
  \item \texttt{P_2:}
    \begin{itemize}
    \item \texttt{while(TestAndSet(Lock))}
    \item \texttt{// =false}
    \item \texttt{//lock=true}
    \item \texttt{lock = false}
    \end{itemize}
\end{itemize}
Atomicity Requirement Revisited

• Our assumption should now be clear; it was correct
• TSE could be used to enforce atomicity for semaphore implementation
• Disabling of interrupts could be used to enforce atomicity for semaphore implementation
  • How are semaphores implemented in OS/161?
  • Is the semaphore implementation based on block() & wakeup() always busy waiting free?
Software-based Solutions for Synchronization
Implementation Alternatives

• Disabling of interrupts
• Atomic instructions (e.g., TSE, SWAP, …)
• If neither of the above is available, *can the critical section problem still be solved?*
• This comes down to solving the critical section problem in software, i.e., algorithmically.
Model Process to Study Problem

- Our model process for looking at this problem
- 2 processes, \( P_0 \) and \( P_1 \)
- General structure of process \( P_i \) (other process \( P_j \))

```plaintext
   do {
       entry section
       critical section
       exit section
       remainder section
   } while (TRUE);
```
Requirements for Solutions

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

• Assume that each process executes at a non-zero speed
• No assumption concerning relative speed of the $n$ processes.
• For the following algorithms 1 to 3, we assume two processes $P_0$ and $P_1$
Algorithm 1

Shared variables:

\[
\text{int } \text{turn}; \hspace{1em} \text{turn} = 0; \quad \text{// initialization}
\]

\[
\text{turn} == i \Rightarrow P_i \text{ may enter CS}
\]

\[P_i: \]

\[
\text{do } \{
\text{busy wait loop}
\text{while (turn != i) ;} \quad \text{Entry section}
\text{critical section}
\text{turn} = 1 - i; \quad \text{Exit section}
\text{remainder section}
\}	ext{ while (TRUE);}
Algorithm 1

Shared variables:
- int turn;  turn = 0; // initialization
- turn - i ⇒ \( P_i \) may enter CS

\( P_i \):
\[
\begin{align*}
&\text{do } \{ \\
&\quad \text{while (turn } \neq i) ; \\
&\quad \text{critical section} \\
&\quad \quad \text{turn } = j ; \\
&\quad \text{remainder section} \\
&\quad \} \text{ while (TRUE);} \\
\end{align*}
\]

\( P_0 \)
\[
\begin{align*}
&\text{while (0 } \neq 0) \\
\end{align*}
\]

\( P_1 \)
\[
\begin{align*}
&\text{while (0 } \neq 1) ; \\
\end{align*}
\]
\[ \text{P}_0: \quad \text{turn} = 0 \quad \text{P}_1 \]

\[ \text{while} \ (0 \neq 0) \]

\[ \text{Critical section} \]

\[ \text{turn} = 1 \quad \text{while} \ (1 \neq 0) \]

\[ \text{Critical section} \]

\[ \text{Critical section} \]

\[ \text{do \ { \text{while} \ (\text{turn} \neq i) \ ; \ \text{critical section} \quad \text{turn} = j; \ \text{remainder section} \} \text{ while (TRUE)}; \]
Algorithm 1

• Enforces a **strictly alternating pattern** between both processes
  – P0, P1, P0, P1, P0
  – P0, P1, P1 is not possible
• That is **mutual exclusion** is guaranteed
• Progress is not (see previous case)
Algorithm 2

Shared variables
- Boolean flag[2];
- flag [0] = flag [1] = false
- flag [i] = true $\Rightarrow P_i$ ready to enter its critical section

$P_i$

do {
  flag[i] = true;
  while (flag[1 - i]) ;
  critical section
  flag [i] = false;
  remainder section
} while (TRUE);

Entry section

Exit section
flag[0] = false
flag[1] = false

$P_0$

flag[0] = true
while( flag[1] );

$P_1$

flag[1] = true
while( flag[0] );

flag[1] = false
flag[0] = false  \quad flag[1] = false

\begin{align*}
P_0 \\
\quad \text{flag}[0] = \text{true} \\
\quad \text{while}(\text{flag}[1]); \\
\end{align*}

\begin{align*}
P_1 \\
\quad \text{flag}[1] = \text{true} \\
\quad \text{while}((\text{flag}[0]); \\
\end{align*}

• Lacks progress requirement
Algorithm 3

• Combined shared variables of algorithms 1 and 2.

• Process \( P_i \)

\[
\begin{align*}
\text{do} \ { \ ( \text{P0's perspective} } & \text{ \ )} \\
\text{flag \[ i \] := true; \ \ / \ \ "I want to enter CS" } & \text{ \ )} \\
\text{turn = 1 - i; \ \ / \ \ "Let P1 go ahead" } & \text{ \ )} \\
\text{while ( flag \[ 1-i \] and turn == 1 – i ) ; } & \text{ \ )} \\
\text{critical section } & \text{ \ )} \\
\text{flag \[ i \] = false; } & \text{ \ )} \\
\text{remainder section } & \text{ \ )} \\
\text{\} while (TRUE)}; & \text{ \ )}
\end{align*}
\]
flag[0] = false flag[1] = false

P₀

flag[0] = true turn = 1

P₁

flag[1] = true

turn = 0

Depending on scheduling
Decision turn is either 1 or 0

while( flag[1] && turn == 1 );
Bakery Algorithm
(synchronization of n processes)

• Before entering its critical section, process receives a number.

• Holder of the smallest number enters the critical section first (Bakery analogy).

• If processes $P_i$ and $P_j$ receive the same number (“due to scheduling accident ☺”)
  – if $i < j$, then $P_i$ is served first
  – else $P_j$ is served first (based on unique PIDs)

• The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
Bakery Algorithm

• Notation < corresponds to lexicographical order
  – \((a,b)\) is \((\text{ticket } \#, \text{ process id })\)
  – \((a,b) < (c,d)\) if \(a < c\) or if \(a = c\) and \(b < d\)
  – \(\text{max } (a_0,\ldots,a_{n-1})\) is a number, \(k\), such that \(k \geq a_i\)
    for \(i = 0, \ldots, n - 1\)

• Shared data // initialization
  
  boolean choosing[n]; // all false

  int nr [n]; // all 0
High-level Description of Algorithm

• Indicate that you are choosing a number
• Choose a number
  – This may occur concurrently and therefore result in two chosen numbers being equal (i.e., kind of race condition)
• Indicate that you have completed choosing a number
• Select the process with the smallest number to proceed into the critical section
Bakery Algorithm: Process $P_i$

do {
    choosing[i] = true;   // indicate choosing a number
    nr[i] = max( nr[0], nr[1], ..., nr[n – 1] ) + 1;
    choosing[i] = false;   // has chosen a number
    for (j = 0; j < n; j++) { // process with smallest nr.
        while (choosing[j]) ; // wait if $P_j$ chooses a nr
        while ( (nr[j] != 0) && ((nr[j], j) < (nr[i],i))  );
    }
    critical section
        If $P_j$ has a number, check it out
        Is it smaller than my own nr.?
    nr[i] = 0;
    remainder section
} while (TRUE);
Why May Two Numbers Be Equal?

If this happens concurrently both numbers may be equal

\[ Nr[0] = \max(0, \ldots 0) \]
\[ + 1 \]
\[ = 1 \]

\[ Nr[k] = \max(0, \ldots 0) \]
\[ + 1 \]
\[ = 1 \]

Both are equal to 1 at this point.
Bakery Algorithm (without choosing)

do {
    nr[i] = max( nr[0], nr[1], ..., nr[n – 1] ) + 1;
    for (j = 0; j < n; j++) {
        while ( (nr[j] != 0) && ( (nr[j], j ) < ( nr[i],i) ) ) ;
    }
    critical section
    nr[i] = 0;
    remainder section
} while (TRUE);
Problem Case

nr[0] = max(0, ...0) ... nr[1] = max(0, ...0) +1

= 1

Critical section

Does P1 have a smaller number? Both are 1.
Well, break ties by looking at PID (0 & 1, here),
(nr[1],1) < (nr[0],0) // (1,1) < 1,0) - false
therefore enter CS (violation of mutual exclusion)
Adding choosing[i] back in

choosing[0] = true
nr[0] = max(0, ...0) ...

choosing[1] = true
nr[1] = max(0, ...0) +1
= 1

for (…)

while (choosing[i]);

while ( …
   ( (nr[0], 0 ) < ( nr[1],1) ) ) ;
// (1,0) < (1,1) - true
// therefore busy wait

Here, we would have waited for P0 to choose a number.

Then we would have let P0 proceed into its CS first
Binary Semaphore
Two Types of Semaphores

• **Counting semaphore** – integer value can range over an unrestricted domain.

• **Binary semaphore** – integer value can range only between 0 and 1; can be simpler to implement.

• Is a binary semaphore the same as a lock?

• **Can we implement a counting semaphore S as a binary semaphore?**
Implementing $S$ as a Binary Semaphore

- Data structures:
  
  ```
  binary-semaphore S1, S2;
  int C:
  ```

- Initialization:
  
  ```
  S1 = 1
  S2 = 0
  C = initial value of semaphore $S$
  ```
Implementing $S$

Wait(C) operation:

```c
wait(S1);
C--;  
if (C < 0) {
    signal(S1);
    wait(S2);
}    
signal(S1);
```

Signal(C) operation:

```c
wait(S1);
C ++;
if (C <= 0) 
    signal(S2);
else
    signal(S1);
```
Synchronization Mechanisms Summary

- Race conditions
- Semaphores (binary, counting)
  - Enforce mutually exclusive use of resources
  - Enforce arbitrary execution patterns (e.g., sequential or ordering constraints)
  - Enforce synchronization constraints (e.g., full, empty, readers/writers constraint)
- Locks and mutexes
  - Enforce mutually exclusive use of resources, exclusively
- Condition variables
  - Enforce waiting for events and conditions (e.g., value of data)
Synchronization Mechanisms Summary

• Monitors (\& critical region construct)
  – Higher-level synchronization primitives
  – Condition variables introduced in this context
• Disabling of interrupts to enforce atomicity
• Test-and-Set Instruction
• Classical problems
  – Bounded buffer problem
  – Dining Philosophers problem
  – Reader Writers problem (reader priority)
Outlook

• Inter-process communication
• OS Architecture
• Scheduling
• Memory management
• …
Critical Region Construct
Critical Region Construct

- High-level synchronization construct
- A shared variable $v$ of type $T$, is declared as:
  
  \[ v: \text{shared } T \]

- Variable $v$ accessed only inside statement
  
  \[ \text{region } v \text{ when } B \text{ do } S \]
  
  where $B$ is a Boolean expression.

- While statement $S$ is being executed, no other process can access variable $v$. 
Critical Regions

• Regions referring to the same shared variable exclude each other in time.

• When a process tries to execute the region statement, the Boolean expression $B$ is evaluated.
  – If $B$ is true, statement $S$ is executed.
  – If it is false, the process is delayed until $B$ becomes true and no other process is in the region associated with $v$. 
Example – Bounded Buffer

• Shared data:

```c
struct buffer {
    int pool[n];
    int count, in, out;
}
```
Bounded Buffer Producer Process

• Producer process inserts `nextp` into the shared buffer

```c
region buffer when( count < n) {
    pool[in] = nextp;
    in:= (in+1) % n;
    count++;
}
```
Bounded Buffer Consumer Process

- Consumer process removes an item from the shared buffer and puts it in `nextc`

```c
region buffer when (count > 0) {
    nextc = pool[out];
    out = (out+1) % n;
    count--;
}
```