PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

Concurrency II - Semaphores

Notices

- **Assignment 1** due 23:59 **Sunday January 26**
- **Lab 2** due 23:59 **Sunday February 9**
- ~~New Section Times~~
  - ~~Mondays 1:00-2:30pm & **6:30-8:00pm**~~
  - ~~Tuesdays 5:00-6:30pm & **6:00-7:30pm** (has overlap, may be wrong)~~
  - ~~Wednesdays 2:00-3:30pm & **3:30-5:00pm**~~
  - ~~**No sections on Thursdays & Fridays**~~
- **Quiz - Possible Midterm Cancellation**
  - **On Canvas now**
  - **Take it, have your say!**

Today's Lecture

- Recap
- Hardware Support
- Concurrency Primitives
- Semaphores

- Assignment 1 Secret Sauce
- A little Lab 2 Secret Sauce

Concurrency in Operating Systems

- ***"The actual or apparent simultaneous execution of threads"***
  - Actual = multicore and/or multiprocessors
  - Apparent = time slicing
  - Today's Reality = both
- Threads running concurrently in the same process typically want to access shared data, but sharing data in an uncontrolled fashion allows that shared data to become:
  - Inconsistent
  - Incorrect
  - Invalid
  - Generally messed up ☹

Critical Sections

- Formalism:
  - **Entry Section**
  - **Critical Section**
  - **Exit Section**
- Requirements:
  - **Mutual Exclusion**
  - **Progress**
  - **Bounded Wait**
- Advice:
  - Read the textbook and review previous lecture handouts for details
  - This is crucial, fundamental information; you MUST know and understand it

Hardware Assistance

- **Problem**: Software solutions are:
  - Complicated
  - NOT atomic
  - NOT generalizable to n-threads
- **Solution**: Provide synchronization primitives
  - Machine instructions (hardware/microcode)
  - System calls (aid programmers)
    - The OS needs to help, as SW at user level is not enough!

Test and Set

- Sets boolean lock variable
- Returns original value
- Implemented in hardware
- Is atomic
- C *equivalent*: (not actually implemented in C, it wouldn't be atomic if it was)

```
bool testAndSet(bool *lock) {
    bool tmp = *lock;
    *lock = true;
    return tmp;
}
```


Using Test and Set

```
bool testAndSet(bool *lock) {
    bool tmp = *lock;
    *lock = true;
    return tmp;
}
```

Shared variables:

```
bool lock = false;
```

Thread n:

```
while (testAndSet(&lock)) {
  yield();
}
// Critical Section
...
lock = false;
// Non-critical
...
```

lock	Set	Action
F	T	Proceed
T	T	Wait

Swap

- Exchanges two values
  - One private, one shared
- Implemented in hardware
- Is atomic
- C *equivalent*: (again, not actually implemented in C)

```
void swap(bool *a, bool *b) {
   bool tmp = *a;
   *a = *b;
   *b = tmp;
}
```


Using Swap

```
void swap(bool *a, bool *b) {
    bool tmp = *a;
    *a = *b;
    *b = tmp
}
```

Shared variables:

```
bool lock = false;
```

Thread n:

```
bool key = true;
while (key) {
  swap(&lock, &key))
}
// Critical Section
...
lock = false;
// Non-critical
...
```

key	lock	key’	lock’	
T	F	F	T	Proceed
T	T	T	T	Wait

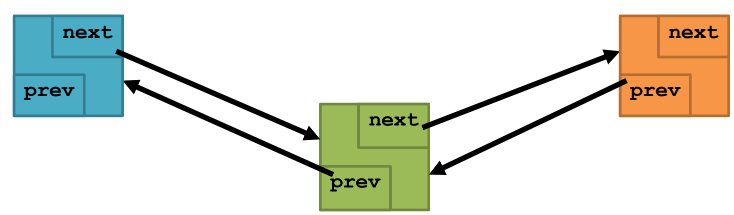
Meet the Requirements?

- **Test And Set** & **Swap** both satisfy
  - Mutual exclusion
  - Progress
- **Bounded Wait** is not satisfied
  - But with some effort we can add additional structures to ensure this
- Unfortunately…
  - Chip manufacturers don’t always include such instructions ☹

- ***This lecture we’ll look at ways the OS supports safe concurrent execution amongst threads***

Concurrency Primitives

- **Concurrency Primitives**: Mechanisms provided by the OS to allow threads to safely work concurrently
- Inserting into a doubly-linked list is a **classic example** of the need for concurrency primitives

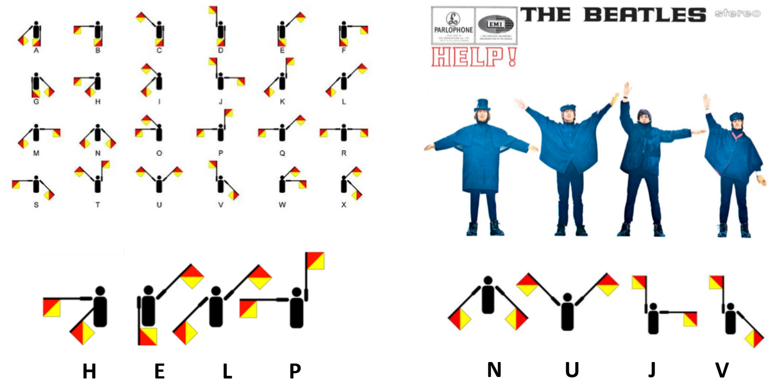


Semaphores

- Dijkstra* recognized the need to manage concurrency amongst threads and processes in 1962**
- But hardware support was far from universal
- In place of machine instructions he proposed the **semaphore**

* Edsger W. Dijkstra 1930 - 2002 : Dutch computer scientist and an early pioneer in many research areas of computing science.
** http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF

Flag Semaphore



"NUJV" ?

Semaphores (Dijkstra -1965)

- P() or wait() **"Proberen"** (test) a.k.a. "down"
- V() or signal() **"Verhogen"** (increment) a.k.a. "up"
- P and V are both atomic
- *Pseudo* code:

```
P(int mutex) {                V(int mutex) {
  while(mutex <= 0);            mutex++;
  mutex--;                    }
}
```

- What's a mutex? MUTual EXclusion

Using Semaphores

Remember:

"down"
```
P(int mutex) {
    while (mutex <= 0);
    mutex--;
}
```

"up"
```
V(int mutex) {
    mutex++;
}
```

Shared variables:

```
int mutex = 1;
```

Thread n:

```
...
...          // Non-critical
P(mutex);    // Entry Section
...          // Critical Section starts
...
...          // Critical Section ends
V(mutex);    // Exit Section
...          // Non-critical again
...
...
```

Counting Semaphores

- The semaphore can also be used to allow `N` threads into a critical section
- Initialize `mutex` to `N`
- First N threads decrement mutex until `mutex == 0`
- Additional threads (> `N`) must wait
- Ideal for controlling access to a fixed number of resources

Binary Semaphores

- A binary semaphore (a "lock") limits mutex values to 0 or 1
- Enforces exclusive entry into critical sections
- **Problem**: we are at the mercy of balanced **P()** and **V()** calls
  – i.e. sloppy programmers can mess things up ☹

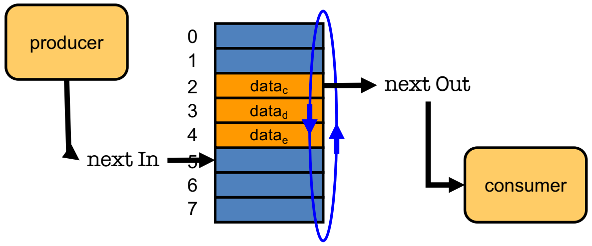
Semaphore Limitations

- Semaphores are useful and general tools
- But they do have drawbacks:
  – Programmers must use them correctly
  – Cannot test busy without blocking
  – We can't simultaneously wait on any of several semaphores
  – Indefinite blocking
  – Cooperative only

Producer / Consumer

- Buffering a potentially infinite amount of data into limited storage (memory) is a standard programming problem
- **Producers** (of the data)
  – Devices, networks, processes
- **Consumers** (of the data)
  – User processes / threads
- Two models:
  – Synchronous
    - Direct "hand over" of data
  – Asynchronous
    - Buffered

Circular Bounded Buffers



put(data$_f$)

get() returns data$_c$

Semaphore Synchronisation

Consider a solution to the **producer-consumer** problem using counting semaphores to allocate slots in a circular buffer of integers of size N …

```
Binary Semaphores:     mutex = 1;
Counting Semaphores:   vacant = N, occupied = 0;   // no. of slots in each state
Buffer Indexes:        nextIn = 0, nextOut = 0;
```

```
void put(int x) {
  P(vacant); // are there some vacant slots?
  P(mutex); // am I allowed to change the buffer?
  buffer[nextIn] = x;
  nextIn = (nextIn++) % N;
  V(mutex); // allow others to change the buffer
  V(occupied); // increment the no. of used slots
}
```

```
int get() {
  P(occupied); // is there at least one piece of data?
  P(mutex); // am I allowed to change the buffer?
  int x = buffer[nextOut];
  nextOut = (nextOut++) % N;
  V(mutex); // allow others to change the buffer
  V(vacant); // increment the no. of available slots
  return x;
}
```


POSIX Semaphores

`sem_init`	Initialise an unnamed semaphore
`sem_open`	Initialise and open a named semaphore
`sem_wait`	Lock a semaphore (put it down, Dijkstra's P)
`sem_post`	Unlock a semaphore (put it up, Dijkstra's V)
`sem_getvalue`	Get the value of a semaphore
`sem_close`	Close a named semaphore
`sem_destroy`	Destroy an unnamed semaphore


```
/*
 * buffer-semaphore.c
 *
 * Circular Bounded Buffer protected by unnamed POSIX semaphores.
 *
 * Copyright (C) 2015-2020 David C. Harrison. All rights reserved.
 *
 * You may not use, distribute, publish, or modify this code without the
 * express written permission of the copyright holder.
 *
 * Producers are fast, consumers are slow. More producers than consumers
 * requires producers to wait (block) for slots to become available.
 *
 * To compile:
 *   gcc -o buffer-semaphore buffer-semaphore.c -lpthread -Wall
 *
 * To run:
 *   ./buffer-semaphore
 *
 * Only works on UNIX and UNIX-like systems, so Linux and macOS are fine.
 *
 * If you're on Windows, you have my condolences. ¯\_(ツ)_/¯
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <semaphore.h>
#include <pthread.h>
#include <uninsd.h>

#define PRODUCERS   2
#define CONSUMERS   5

#define MAX_SLEEP_MILLISECONDS  3000
#define MAX_PUTS_PER_PRODUCER  1000

#define BUFFER_SIZE 10

typedef struct buffer_t
{
   int buf[BUFFER_SIZE];      /* shared buffer */
   int next_in;               /* next slot to add an element to (may not currently be vacant) */
   int next_out;              /* next slot to get an element from (may not currently be occupied) */
   sem_t occupied;            /* counting semaphore for occupied slots */
   sem_t vacant;              /* counting semaphore for vacant slots */
   sem_t mutex;               /* binary semaphore to facilitate mutually exclusive access to the buffer */
}
Buffer;

Buffer buffer;

static void sleep_ms(int msec) {
   struct timespec ts;
   ts.tv_sec = msec / 1000;
   ts.tv_nsec = (msec % 1000) * 1000000;
   nanosleep(&ts, NULL);
}
```



```
/* Initialise the circular bounded buffer BUFFER. */
void buffer_init(Buffer *buffer) {
   sem_init(&buffer->occupied , 0 , 0);
   sem_init(&buffer->vacant , 0, BUFFER_SIZE);
   sem_init(&buffer->mutex , 0, 1);
}

/* Blocking insert of VALUE into BUFFER. */
void buffer_put(Buffer *buffer, int value) {
   sem_wait(buffer->vacant);    // wait until at least one vacant slot
   sem_wait(buffer->mutex);     // wait for exclusive access to the shared buffer

   buffer->buf[buffer->next_in] = value;
   buffer->next_in = (buffer->next_in+1) % BUFFER_SIZE;

   sem_post(buffer->mutex);     // release shared buffer so other threads can use it
   sem_post(buffer->occupied); // decrement the number of occupied slots
}

/* Blocking retrieval of next BUFFER entry */
int buffer_get(Buffer *buffer) {
   sem_wait(buffer->occupied); // wait until at least one slot has a valid entry
   sem_wait(buffer->mutex);     // wait for exclusive access to the shared buffer

   int value = buffer->buf[buffer->next_out];
   buffer->next_out = (buffer->next_out+1) % BUFFER_SIZE;

   sem_post(buffer->mutex);     // release the shared buffer so other threads can use it
   sem_post(buffer->vacant);    // decrement the number of vacant slots

   return value;
}
```



```
static void *produce(void *arg) {
   long tid = (long) arg;
   int value;
   printf("P %ld running\n", tid);

   for (int i = 0; i < MAX_PUTS_PER_PRODUCER; i++) {
      value = tid*1000+i;
      printf("=> P %ld produced %04d\n", tid, value);
      buffer_put(&buffer, value);
   }

   printf("P %ld ### finished ###\n", tid);
   pthread_exit(NULL);
   return NULL;
}

static void *consume(void *arg) {
   long tid = (long) arg;
   int value, millis;
   printf("C %ld running\n", tid);

   for (;;) {
      millis = rand() % MAX_SLEEP_MILLISECONDS;
      value = buffer_get(&buffer);
      printf("<= C %ld consumed %04d - sleeping for %dms\n", tid, value, millis);
      sleep_ms(millis);
   }

   // Will never return, consumers are in an infinite loop
   return NULL;
}
```



```
// Reminder of some literals defined earlier
#define PRODUCERS   2
#define CONSUMERS   5
// End reminder

int main(int argc, char* argv[])
{
   buffer_init(&buffer);
   pthread_t junk;

   for(long tid = 0; tid < CONSUMERS; tid++) {
      pthread_create(&junk, NULL, consume, (void*)tid);
   }

   for (long tid = 0; tid < PRODUCERS; tid++) {
      pthread_create(&junk, NULL, produce, (void*)tid);
   }

   pthread_exit(NULL);

   // Will never reach here as consumers run forever
   return (EXIT_SUCCESS);
}
```

```
$ ./buffer-semaphore
C 0 running
C 2 running
C 1 running
C 3 running
C 4 running
P 0 running
P 1 running
=> P 0 produced 0000
=> P 1 produced 1000
=> P 0 produced 0001
<= C 0 consumed 0000 - sleeping for 1807ms
=> P 1 produced 1001
<= C 2 consumed 1000 - sleeping for 1249ms
=> P 0 produced 0002
<= C 1 consumed 0001 - sleeping for 1073ms
=> P 1 produced 1002
<= C 3 consumed 1001 - sleeping for 1658ms
=> P 0 produced 0003
<= C 4 consumed 0002 - sleeping for 1930ms
=> P 1 produced 1003
=> P 0 produced 0004
=> P 1 produced 1004
=> P 0 produced 0005
=> P 1 produced 1005
=> P 0 produced 0006
=> P 1 produced 1006
=> P 0 produced 0007
=> P 1 produced 1007
=> P 0 produced 0008
<= C 1 consumed 1002 - sleeping for 272ms
=> P 1 produced 1008
<= C 2 consumed 0003 - sleeping for 2544ms
=> P 0 produced 0009
<= C 1 consumed 1003 - sleeping for 878ms
=> P 1 produced 1009
...
...
...
```

```
...
...
...
<= C 1 consumed 0989 - sleeping for 652ms
=> P 0 produced 0995
<= C 3 consumed 1990 - sleeping for 1059ms
=> P 1 produced 1996
<= C 2 consumed 0990 - sleeping for 2577ms
=> P 0 produced 0996
<= C 4 consumed 1991 - sleeping for 1670ms
=> P 1 produced 1997
<= C 1 consumed 0991 - sleeping for 151ms
=> P 0 produced 0997
<= C 1 consumed 1992 - sleeping for 224ms
=> P 1 produced 1998
<= C 1 consumed 0992 - sleeping for 2084ms
=> P 0 produced 0998
<= C 3 consumed 1993 - sleeping for 1020ms
=> P 1 produced 1999
<= C 0 consumed 0993 - sleeping for 1015ms
=> P 0 produced 0999
<= C 3 consumed 1994 - sleeping for 1219ms
P 1 ### finished ###
<= C 4 consumed 0994 - sleeping for 2177ms
P 0 ### finished ###
<= C 0 consumed 1995 - sleeping for 695ms
<= C 2 consumed 0995 - sleeping for 1895ms
<= C 1 consumed 1996 - sleeping for 1664ms
<= C 3 consumed 0996 - sleeping for 600ms
<= C 0 consumed 1997 - sleeping for 118ms
<= C 0 consumed 0997 - sleeping for 117ms
<= C 0 consumed 1998 - sleeping for 1723ms
<= C 3 consumed 0998 - sleeping for 1617ms
<= C 4 consumed 1999 - sleeping for 2754ms
<= C 1 consumed 0999 - sleeping for 568ms

^C
```


Self Study (totally not graded, just do it for fun!)

- Copy the source code from handouts
  - Build it, run it, study it
- Change it so producers are slow and consumers are fast
  - **Q: What happens?**
- Increase no. of producers & decrease no. of consumers
  - **Q: What happens?**
- Take it to parties in an attempt to impress your friends
  - **Q: What happens?**
  - **A: You no longer have any friends.**

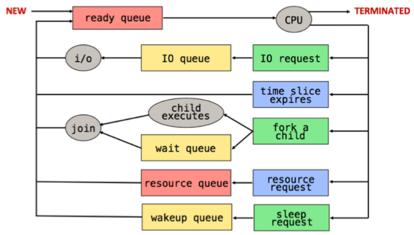
Assignment 1 - Secret Sauce

- Using POSIX Shared Memory System Calls
  - Re-install and add `-lrt` flag to `Makefile.libs`
- When a process calls `fork()`
  - How similar are the parent and child processes?
  - Do they share a program counter?
  - When does the program counter of each change?
  - Do they share executable code?
  - What happens when they each arrive at a return statement?
- Watch your processes!!
  - Two users where chewing up 64,000 processes between them ☹
  - See how many processes you have and kill 'em:

```
$ ps aux | grep <cruzid>
$ killall -u <cruzid>
```


Lab 2 - Secret Sauce

- **We have not yet covered in lectures everything you need to understand, but there's a whole bunch of stuff we do know:**
- Pintos is a time-sharing OS
- When a **time-slice** expires:
  - The running (current) thread is taken off the CPU and placed at the back of the **ready queue**
  - The thread at the front of the ready queue is then removed and placed on the CPU



- If we want to implement **priority based scheduling** we have to sort the ready queue as and when appropriate

Next Lecture

- Condition Variables