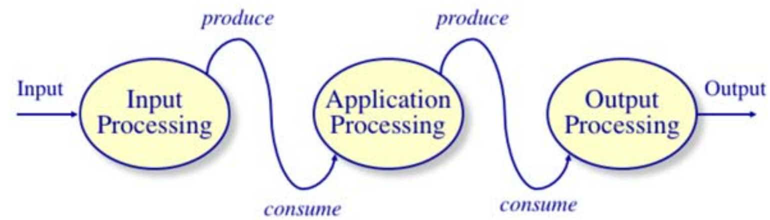
COMP 530
Introduction to Operating Systems

Notes on HW3
Implementing Producer/Consumer systems

Kevin Jeffay
Department of Computer Science
University of North Carolina at Chapel Hill
jeffay@cs.unc.edu
September 20, 2017

http://www.cs.unc.edu/~jeffay/courses/comp530

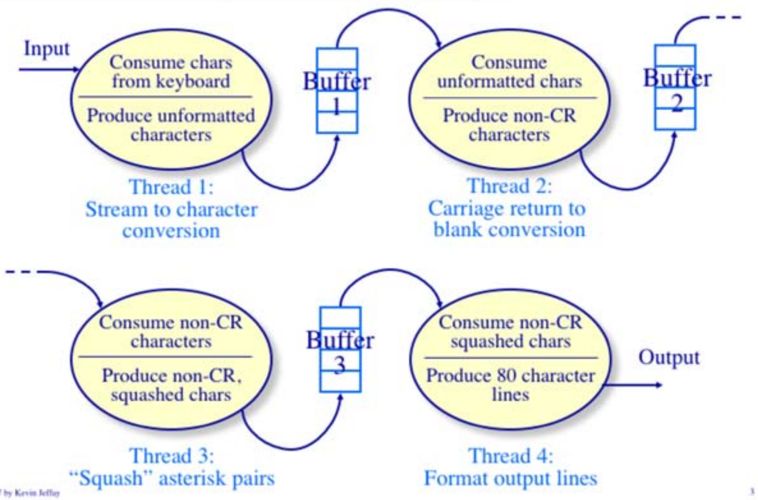
HW 3
A multi-threaded producer/consumer system



- Re-implement your solution to HW 1 as a pipeline of C threads
  - » Use the *ST* ("State Threads") open source thread package for C
  - » Implement a sequence of producer/consumer systems using *ST* threads and semaphores

HW 3

A multi-threaded producer/consumer system



 3

HW 3

Details

- Use semaphores to synchronize producer/consumer pairs

```
/* semaphore.h
 *
 * An ST extension implementing classical general/counting semaphores
 *
 * Public Interface -
 *
 *  typedef semaphore
 *
 *  void down(semaphore* s) - blocks the calling thread if the
 *                            value of the of the semaphore is 0,
 *                            otherwise the value of the semaphore
 *                            is decremented.
 *
 *  void up(semaphore* s)   - increments the value of the semaphore.
 *                            If a thread is blocked on the semaphore
 *                            it is woken up.
 *
 *  void createSem(semaphore* s, int value ) - sets the initial
 *                            integer value of the semaphore. 'value'
 *                            must be nonnegative.
 */
```

 4

HW 3 Notes

Asserting synchronization conditions

```
globals
fullBuffers  : semaphore := 0      buf   : array [0..n-1] of char
emptyBuffers : semaphore := n      nextIn,nextOut : 0..n-1 := 0
```

```
process Producer
begin
  loop
    <produce a character "c">

    emptyBuffers.down()
    <assert that the awaited
      condition is true>

    buf[nextIn] := c
    nextIn := nextIn+1 mod n

    <assert that the synchronization
      condition is true>
    fullBuffers.up()
    <assert ??>
  end loop
end Producer
```

```
process Consumer
begin
  loop
    fullBuffers.down()
    <assert that the awaited
      condition is true>

    c := buf[nextOut]
    nextOut := nextOut+1 mod n

    <assert that the synchronization
      condition is true>
    emptyBuffers.up()

    <consume a character "c">
  end loop
end Consumer
```

 5

HW 3 Notes

Asserting synchronization conditions

```
globals
fullBuffers  : semaphore := 0      buf   : array [0..n-1] of char
emptyBuffers : semaphore := n      nextIn,nextOut : 0..n-1 := 0
```

```
process Producer
begin
  loop
    <produce a character "c">

    emptyBuffers.down()
    <assert that the awaited
      condition is true>

    buf[nextIn] := c
    nextIn := nextIn+1 mod n

    <assert that the synchronization
      condition is true>
    fullBuffers.up()
    <assert ??>
  end loop
end Producer
```

- In C there's a simple function for stating assertions:

```
#include<assert.h>

void foo(char* string)
 {
    assert(string != NULL);
      :
      :
 }
```

 6

HW 3 Notes

Software engineering issues

```
globals
  fullBuffers  : semaphore := 0    buf   : array [0..n-1] of char
  emptyBuffers : semaphore := n    nextIn,nextOut : 0..n-1 := 0
```

```
process Producer
begin
  loop
    <produce a character "c">

            emptyBuffers.down()
            buf[nextIn] := c
  deposit   nextIn := nextIn+1 mod n
            fullBuffers.up()
  end loop
end Producer
```

```
process Consumer
begin
  loop
            fullBuffers.down()
            c := buf[nextOut]
  remove    nextOut := nextOut+1 mod n
            emptyBuffers.up()

    <consume a character "c">
  end loop
end Consumer
```

- Modularity and information hiding principles dictate that all the buffer management should be part of an abstract buffer class

HW 3 Notes

Software engineering issues

```
globals
  fullBuffers  : semaphore := 0    buf   : array [0..n-1] of char
  emptyBuffers : semaphore := n    nextIn,nextOut : 0..n-1 := 0
```

```
process Producer
begin
  loop
    <produce a character "c">

            emptyBuffers.down()
            buf[nextIn] := c
  deposit   nextIn := nextIn+1 mod n
            fullBuffers.up()
  end loop
end Producer
```

```
process Consumer
begin
  loop
            fullBuffers.down()
            c := buf[nextOut]
  remove    nextOut := nextOut+1 mod n
            emptyBuffers.up()

    <consume a character "c">
  end loop
end Consumer
```

- Construct a bounded buffer abstract data type with functions *deposit* and *remoove*
  - » Implement your buffer data type in a separate file called *buffer.c* (with definitions in the file *buffer.h*)

HW 3
Grading

- Your program should behave *exactly* as a correct version of HW1 would
  - » Your HW3 program will be tested the same way your HW1 program was tested
  - » Thus, you can use your HW1 program to evaluate the correctness of HW3
- For extra credit, use the *condition variable* structures in *ST* to implement your own version of semaphores
  - » Put your semaphore implementation in files *semaphore.c* and *semaphore.h*
  - » DON'T ATTEMPT THE EXTRA CREDIT UNTIL YOU HAVE A FULLY FUNCTIONING SOLUTION TO THE "BASE" ASSIGNMENT!!