CS 134:
Operating Systems
Computer Hardware
Synchronization
Overview

Administrivia

Hardware
  Overview
  I/O Hardware
  Interrupts

Synchronization
  Critical Sections
  Hardware Support
  Higher-Level Mechanisms

Dining Philosophers
Status: I’m working on it; initial setup takes a bit of time
  ▶ So when I post homework, plan for that time!

First assignment will be “get going”

You should have your group formed by now
Activities to do before Thursday:

- Find out about system calls
  - Read manual pages on `getpid`, `stime`, `readdir`
  - About how many system calls does Linux have? (Hint: manual pages live in `/usr/share/man`)
  - Run `strace` (on Knuth or other Linux) on a simple program such as `true`, `echo`, or `ls`
Need to perform computation!

- Memory contains program instructions and program data
- Processor registers maintain processor state. Registers include:
  - General purpose (address & data) registers
  - Instruction pointer (aka program counter)
  - Stack pointer(s)
  - Control and status registers
Computer Hardware—I/O Devices

Need to communicate with the world!

- I/O devices and CPU execute concurrently
- Devices have hardware controllers
  - Handles devices of a particular device type
  - Some level of autonomy
  - Local buffer
- I/O is from the device to local buffer of controller
Programmed I/O

After I/O starts, control returns to user program only on I/O completion
- CPU waits until I/O completes.
- At most one I/O request is outstanding at a time
  - No simultaneous I/O processing
Polling == Querying the I/O device
Separate I/O into two parts:

- Initiation
- Polling

Advantages?
Separate I/O into two parts:

- Initiation
- Asynchronous notification
User-level code almost always uses “programmed I/O” (e.g. read and write on a file)

Why?
CPU needs another feature...
Handling an Interrupt

What needs to happen:

▶ Save state
  ▶ All registers
  ▶ Switch stacks?

▶ Find out what interrupt was . . .
  ▶ Polling
  ▶ Vectored interrupts
Various types

- Software exception (also called a trap)
- Timer
- I/O
- Hardware failure

A modern operating system is *interrupt driven*
We’ve covered interrupts, but hardware has other cool features, including:

- Caches
- Memory management
- Protection

We’ll come back to hardware as we address these topics.
Recap

Solution to I/O waiting was:

▶ Do something else during I/O!

But doing two (or more) things at once introduces headaches!
Solution to I/O waiting was:

- Do something else during I/O!

But doing two (or more) things at once introduces headaches!
Uncontrolled access to shared data

⇒ Race conditions
Two threads:

- **Producer**: Creates data items
- **Consumer**: Uses them up

We’ll look at the problem using a shared array...
enum { N = 128}; // maximum buffer capacity
volatile item buffer[N]; // the buffer itself
volatile int in = 0; // buffer in cursor (moved by producer)
volatile int out = 0; // buffer out cursor (moved by consumer)

void producer() {
    item made_item;
    for (; ; ) {
        made_item = make_item();
        while ((in + 1) % N == out) {
            /* buffer full---wait */
        }
        buffer[in] = made_item;
        in = (in + 1) % N;
    }
}

void consumer() {
    item usable_item;
    for (; ; ) {
        while (in == out) {
            /* buffer empty---wait */
        }
        usable_item = buffer[out];
        out = (out + 1) % N;
        use_item(usable_item);
    }
}
Okay?

enum { N = 128 }; // maximum capacity of the buffer
volatile item buffer[N]; // the buffer itself
volatile int count = 0; // how many things are in the buffer

void producer() {
    int in = 0;
    item made_item;

    for (;;) {
        made_item = make_item();
        while (count == N) {
            /* buffer full---wait */
        }
        buffer[in] = made_item;
        in = (in + 1) % N;
        ++count;
    }
}

void consumer() {
    int out = 0;
    item usable_item;

    for (;;) {
        while (count == 0) {
            /* buffer empty---wait */
        }
        usable_item = buffer[out];
        out = (out + 1) % N;
        --count;
        use_item(usable_item);
    }
}
The MIPS code for \texttt{++count} is as follows

\begin{verbatim}
lw $2, count
nop
addu $2, $2, 1
sw $2, count
\end{verbatim}
The critical section problem exists where $n > 1$ processes all compete to use some shared data

- But not always—certain other conditions apply
  - Roughly, different processes see conflicting data
- Code that accesses shared data = **critical section**
- Must ensure mutual exclusion for critical sections

**Generic Example:**

```c
/* Shared data... */

void foo()
{
    for (; ; ) {
        /* enter critical section */
        foo_cs_actions();
        /* leave critical section */
        foo_other_actions();
    }
}

void bar()
{
    for (; ; ) {
        /* enter critical section */
        bar_cs_actions();
        /* leave critical section */
        bar_other_actions();
    }
}
```
Critical-Section Problem—Solution Requirements

Must satisfy the following requirements:

- **Mutual Exclusion**
- **Progress**
- **Bounded Waiting** (also known as **No Starvation**)

(Assume processes don’t hang/die inside the critical section.)

(Can’t assume anything about execution speeds or number of CPUs.)

Mutual exclusion: If a process is executing in its critical section, then no other processes can be executing in their critical sections.

Progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter its critical section next cannot be postponed indefinitely.

Bounded waiting: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has asked to enter its critical section.
/* Shared data---Whose turn it is */
volatile enum { Foo, Bar } turn = Foo;

void foo()
{
  for ( ; ; ) {
    while (turn != Foo) {
      /* let bar take its turn */
    }
    foo_cs_actions();
    turn = Bar;
    foo_other_actions();
  }
}

void bar()
{
  for ( ; ; ) {
    while (turn != Bar) {
      /* let foo take its turn */
    }
    bar_cs_actions();
    turn = Foo;
    bar_other_actions();
  }
}

Does this code satisfy our requirements?
Critical-Section Problem—Solution?

/* Shared data---Who is busy? */
volatile bool foo_busy = false;
volatile bool bar_busy = false;

void foo()
{
    for ( ; ; ) {
        foo_busy = true;
        while (bar_busy == true) {
            /* let bar finish */
        }
        foo_cs_actions();
        foo_busy = false;
        foo_other_actions();
    }
}

void bar()
{
    for ( ; ; ) {
        bar_busy = true;
        while (foo_busy == true) {
            /* let foo finish */
        }
        bar_cs_actions();
        bar_busy = false;
        bar_other_actions();
    }
}
How about this version?

```c
void task(const int i)
{
    for ( ; ; ) {
        splhigh();
        cs_actions(i);
        spl0();
        other_actions(i);
    }
}
```
Critical-Section Problem—Solution?

Or this one?

```c
/* Shared data */
bool lock = false; // shared mutual exclusion lock

void task(const int i)
{
    for ( ; ; ) {
        while (test_and_set(lock)) {
/* do nothing---wait for lock to be released */
        }
    cs_actions(i);
    lock = false;
    other_actions(i);
    }
}
```
You've seen 'em in 105:

```c
void task(const int i) {
    for ( ; ; ) {
        P(oursem);
        cs_actions(i);
        V(oursem);
        other_actions(i);
    }
}
```
Semaphores

Two fundamental operations

\textbf{P} \textit{proberen} \hspace{5mm} \text{down} \hspace{5mm} \text{dec} \hspace{5mm} \text{wait} \hspace{5mm} \text{Try to grab the semaphore}

\textbf{V} \textit{verhogen} \hspace{5mm} \text{up} \hspace{5mm} \text{inc} \hspace{5mm} \text{signal} \hspace{5mm} \text{Release the semaphore}

Semaphores have an associated count!

\begin{itemize}
  \item \textbf{P} -- Sleep until count is nonzero; once positive, decrement count
  \item \textbf{V} -- Increment count, wake any sleepers
\end{itemize}
Semaphores

Two fundamental operations

P proberen down dec wait Try to grab the semaphore
V verhogen up inc signal Release the semaphore

Semaphores have an associated count!

- P—Sleep until count is nonzero; once positive, decrement count
- V—Increment count, wake any sleepers
enum { N = 128 }; // maximum capacity of the buffer
volatile item buffer[N]; // the buffer itself
struct sem *empty_slot; // any free slots? (initialized to N)
struct sem *filled_slot; // any filled slots? (initialized to 0)

void producer() {
    int in = 0;
    item made_item;

    while (true) {
        made_item = make_item();
        P(empty_slot)
        buffer[in] = made_item;
        in = (in + 1) % N;
        V(filled_slot);
    }
}

void consumer() {
    int out = 0;
    item usable_item;

    while (true) {
        P(filled_slot);
        usable_item = buffer[out];
        out = (out + 1) % N;
        V(empty_slot);
        use_item(usable_item);
    }
}
Bounded Buffer with Semaphores

enum { N = 128 }; // maximum capacity of the buffer
item_queue buffer; // the buffer itself
struct sem *empty_slot; // any free slots? (initialized to N)
struct sem *filled_slot; // any filled slots? (initialized to 0)
struct sem *mutex; // protection for the buffer (initialized to 1)

void producer()
{
    item made_item;
    for ( ; ; ) {
        made_item = make_item();
P(empty_slot)
P(mutex);
        put_item(buffer, made_item);
        V(mutex);
        V(filled_slot);
    }
}

void consumer()
{
    item usable_item;
    for ( ; ; ) {
        P(filled_slot);
P(mutex);
        usable_item = get_item(buffer);
        V(mutex);
        V(empty_slot);
        use_item(usable_item);
    }
}
Each philosopher alternates between periods of
▶ Thinking
▶ Eating

Each philosopher
▶ Shares chopsticks with neighbors
▶ Must not starve

This slide has animations
Each philosopher alternates between periods of
- Thinking
- Eating

Each philosopher
- Shares chopsticks with neighbors
- Must not starve

Philosophers also must not deadlock
Dining Philosophers

enum { N = 5 };  // five philosophers
enum { HUNGRY, THINKING, EATING } state[N];  // everyone’s state
struct sem mutex = 1;  // mutual exclusion for critical regions
struct sem s[N];  // one semaphore per philosopher

void philosopher(int i)
{
    for (; ; ) {
        think();  // philosopher is thinking
        take_chopsticks(i);  // acquire chopsticks (block if need be)
        eat();  // yum-yum
        put_chopsticks(i);
    }
}

void test(int i)
{
    if (state[i] == HUNGRY && state[left(i)] != EATING && state[right(i)] != EATING) {
        state[i] = EATING;
        V(s[i]);  // let philosopher i eat!
    }
}
void take_chopsticks(int i) {
    P(mutex);  // enter critical region
    state[i] = HUNGRY;
    test(i);  // try to acquire 2 chopsticks
    V(mutex);  // exit critical region
    P(s[i]);  // block if chopsticks were not acquired
}

void put_chopsticks(int i) {
    P(mutex);  // enter critical region
    state[i] = THINKING;
    test(left(i));  // see if left neighbor can now eat
    test(right(i));  // see if right neighbor can now eat
    V(mutex);  // exit critical region
}