Module II

Programming Models And Concurrent Processing

The Three Basic Programming Models

- Synchronous event loop
  - Used only in low-end systems
  - Programmer writes a loop that handles multiple activities
- Asynchronous event handlers
  - Programmer writes a function for each “event”
  - Often associated with graphical interface
  - Example *mouse-over* event

The Three Basic Programming Models
(continued)

- Concurrent Processing
  - Fundamental concept that dominates OS design
  - The model you have been using
  - Programmer writes separate applications programs and the operating system allows them to run at the same time

Real Vs. Apparent Concurrency

- *Real concurrency* is only achieved when hardware operates in parallel
  - I/O devices operate at same time as the processor
  - Multiple processors/cores each operate at the same time
- *Apparent concurrency* is achieved with *multitasking* (aka *multiprogramming*)
  - The most fundamental role of an operating system
  - Multiple programs appear to operate simultaneously

In The Concurrent Model

- User(s) start multiple computations running at any time
- The OS switches processor(s) (i.e., core(s)) among available computations quickly
- To a human, all computations appear to proceed in parallel

Two Basic Categories Of Operating Systems

- Timesharing operating system
  - The operating system gives the processor to a computation for a short time (e.g., a millisecond) and then moves the processor to another
  - As the user starts more computations, each receives less of the processor's time (i.e., they appear to run slower)
- Real-time operating system
  - Often used in embedded devices (e.g., a smart phone)
  - User can specify priorities for processes (e.g., a phone call has priority over a music player)

Terminology

- A *program* consists of static code and data
- A *function* is a unit of application program code
- A *process* (also called a *thread of execution*) is an active computation (i.e., the execution or “running” of a program)

A Process

- Is an OS abstraction
- Can be created when needed (an OS system call allows a running process to create a new process)
- Is managed entirely by the OS and is unknown to the hardware
- Operates concurrently with other processes

Example Of Process Creation In Xinu (Part 1)

```
/* ex2.c - main, sndA, sndB */

#include <xinu.h>

/*------------------------------------------------------------------------
 * sndA  -  Repeatedly emit 'A' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndA(void)
{
        while( 1 )
                putc(CONSOLE, 'A');
}

/*------------------------------------------------------------------------
 * sndB  -  Repeatedly emit 'B' on the console without terminating
 *------------------------------------------------------------------------
 */
void    sndB(void)
{
        while( 1 )
                putc(CONSOLE, 'B');
}
```

Example Of Process Creation In Xinu (Part 2)

```
/*------------------------------------------------------------------------
 * main  -  Example of creating processes in Xinu
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndA, 1024, 20, "process 1", 0) );
        resume( create(sndB, 1024, 20, "process 2", 0) );
}
```

The Difference Between Function Call And Process Creation

- A normal function call
  - Only involves a single computation
  - Executes synchronously (caller waits until the call returns)
- The *create* system call
  - Starts a new process and returns
  - Both the old process and the new process proceed to run after the call

The Distinction Between A Program And A Process

- A sequential program is
  - Declared explicitly in the code (e.g., with the name *main*)
  - Is executed by a single thread of control
- A process
  - Is an OS abstractions that is not visible in a programming language
  - Is created independent of code that is executed
  - Important idea: multiple processes can execute the same code concurrently
- In the following example, two processes execute function *sndch* concurrently

Example Of Two Processes Running The Same Code

```
/* ex3.c - main, sndch */

#include <xinu.h>

void    sndch(char);

/*------------------------------------------------------------------------
 * main  -  Example of 2 processes executing the same code concurrently
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(sndch, 1024, 20, "send A", 1, 'A') );
        resume( create(sndch, 1024, 20, "send B", 1, 'B') );
}

/*------------------------------------------------------------------------
 * sndch  -  Output a character on a serial device indefinitely
 *------------------------------------------------------------------------
 */
void    sndch(
          char  ch              /* The character to emit continuously   */
        )
{
        while ( 1 )
                putc(CONSOLE, ch);
}
```

The Point To Note

A program consists of code executed by a single process (i.e., thread of control). In contrast, concurrent processes are not uniquely associated with a piece of code; multiple processes can execute the same code simultaneously.

Storage Allocation When Multiple Processes Execute

- Various memory models exist for concurrent processes
- Each process requires its own storage for
  - A runtime stack of function calls
  - Local variables
  - Copies of arguments passed to functions
- A process *may* have private heap storage as well

Consequence For Programmers

A copy of function arguments and local variables is associated with each process executing a particular function, *not* with the code in which the variables and arguments are declared.

Process Exit

- In Xinu, as in many operating systems. a process can exit in two ways
  - The process can be *killed* (i.e., terminated prematurely)
  - The process can Exit normally

Killing A Xinu Process

- A process with process ID *n* can be terminated by calling

```
kill(n);
```

- The Xinu system call *getpid* returns the process ID of the currently executing process, so a process can kill itself by calling

```
kill(getpid());
```

Normal Process Exit

- A process exits normally by returning from the function in which it started
- To be precise: exit occurs when the process returns from the top function on the activation stack, even if recursive calls occur

Concurrent Processes, Shared Memory, And Race Conditions

- Many operating systems permit processes to share memory
- Example: Xinu makes all global variables shared
- When two or more processes attempt to change a shared variable, errors can arise

Classic Example Of Two Processes Incrementing a Shared Variable

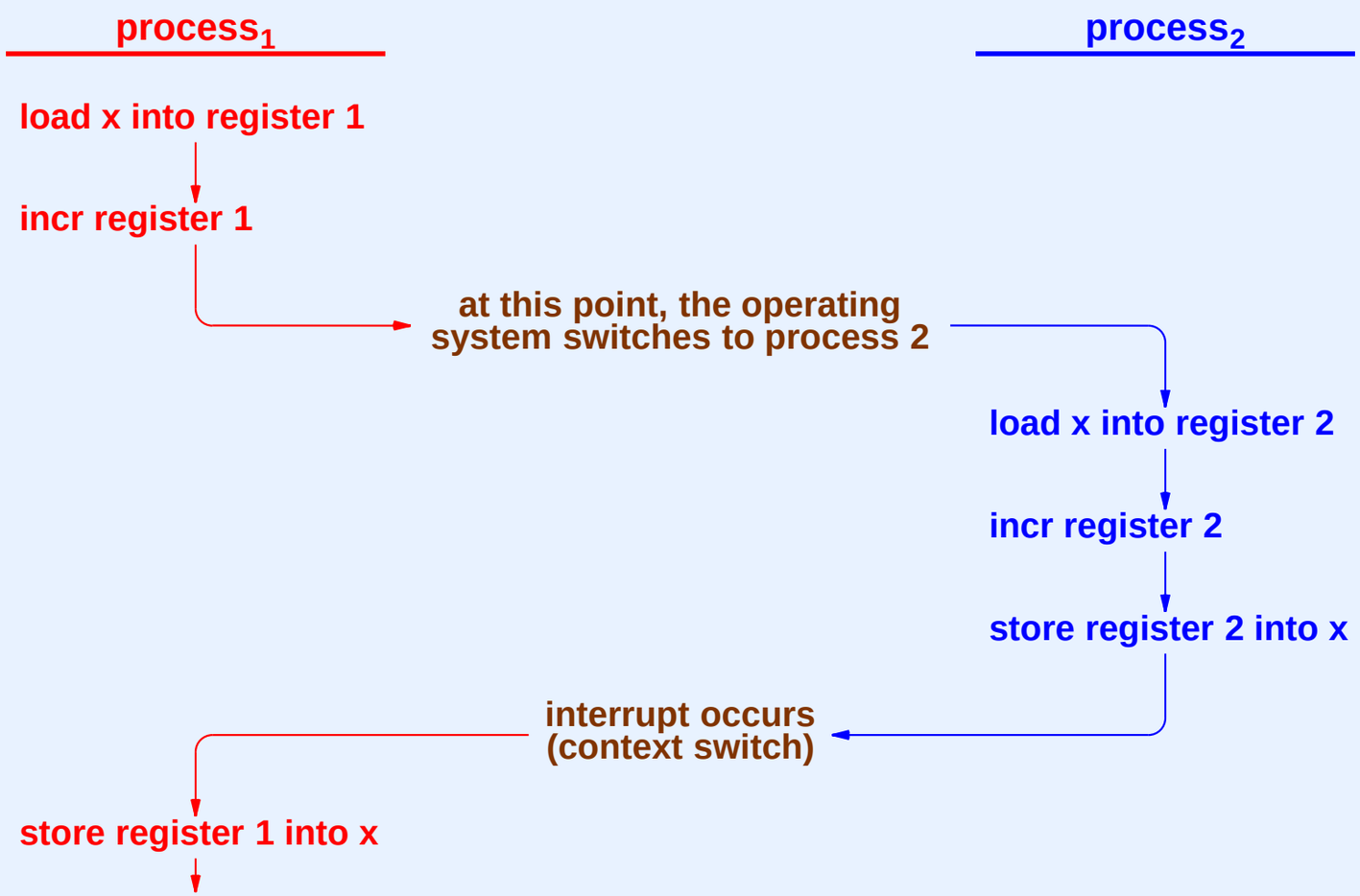
- Suppose integer *x* is shared among all processes
- Consider what happens if two processes each execute *x++;*
- The compiler generates the following instructions for x++
  - Load the value of *x* from memory into a register
  - Increment the register
  - Store the value from the register into *x* in memory
- The point:

Although a programmer thinks of x++ as a single operation, the underlying hardware performs multiple steps to increment an integer.

Classic Example Of Two Processes Incrementing a Shared Variable (continued)

- The operating system may switch back and forth between processes at any time, which means the following could occur when two processes execute x++
  - Process 1 loads the value of *x* into one of its registers and increments its register
  - The OS switches to process 2, which loads the value of *x* into one of its registers, increments the register, and stores the value from the register into *x*
  - The OS switches back to process 1, which stores the value from its register into *x*
- Result: The value of *x* is only incremented by 1 even though two processes executed x++;

Illustration Of Two Processes Incrementing x



Solving Race Conditions: Synchronization

- Operating systems provide mechanisms a programmer can use to guarantee correct results on shared variables
- Example: Xinu uses counting semaphores
- We will learn much more later in the course, but here is a quick preview

Mutual Exclusion

- Allows only one process to access a shared variable at a given time
- Requires a programmer to
  - Create a semaphore with an initial count of 1

```
sid32 mysemaphore;
mysemaphore = semcreate(1);
```

  - Place calls to *wait* and *signal* around each reference to the shared variable, as in:

```
wait(mysemaphore);
x++;
signal(mysemaphore);
```

- The operating system guarantees that only one process will pass the call to *wait* until a call to *signal* occurs; the OS blocks other processes (keeps them from running)

Producer-Consumer Interaction

- Is slightly more complex than merely protecting access to a variable
- One process “produces” something that another process must “consume”
- Example: two processes pass values in a shared integer (initial value 0)
  - A “producer” process contains a loop that increments n 2000 times
  - A “consumer” process contains a loop that prints each of the values
- If the two processes run without synchronization, it is unlikely that the output will list all values of the variable
- Reason: the producer may iterate many times before the consumer process runs or may even finish before the consumer runs at all (all outputs list a value of 2000)

Synchronizing Producer-Consumer Interaction

- Can be done with two semaphores that the processes share
  - A *consumer* semaphore has an initial value of 0
  - A *producer* semaphore has an initial value of 1
- The producer process executes the following 2000 times

```
wait(consumed);
n++;
signal(produced);
```

- The consumer process executes the following 2000 times

```
wait(produced);
printf("%d\n", n);
signal(consumed);
```

- You do not need to understand synchronization yet — just appreciate that an operating system provides easy-to-use synchronization mechanisms

To Learn More

- Read the sections in Chapter 2 on producer-consumer synchronization and mutual exclusion and look at the example code
- Note that a programmer only needs to add a few calls of semaphore functions to enable processes to share variables correctly
- What's coming
  - PSO exercises where you will have a chance to write code that uses semaphore functions to coordinate processes
  - A future module where you will learn how an operating system implements semaphores

Type Names In Xinu

- A type name can specify
  - An abstract meaning that specifies the intended use (e.g., “integer”)
  - A size (especially important in embedded systems)
- Xinu type names specify both, for example:

Type	Meaning
byte	unsigned 8-bit value
bool8	8-bit value used as a Boolean
int16	signed 16-bit integer
uint16	unsigned 16-bit integer
int32	signed 32-bit integer
uint32	unsigned 32-bit integer
sid32	32-bit semaphore ID

A Hint For Working Within An Operating System

- Applications use *putc* or *printf* to print output on the CONSOLE
- Both printf and putc
  - Require the I/O subsystem within the operating system to work correctly
  - Interrupts to be enabled
- In most cases, operating system functions disable interrupts temporarily
- Therefore, programmers must use specialized functions that work in the operating system kernel
  - Use *kprintf* instead of *printf*
  - Use *kputc* instead of *putc*

Questions?