Lecture Overview

- Overview of cooperating processes and synchronization
  - Terms and concepts
  - Mutual exclusion with busy waiting and spin locks
  - Mutual exclusion and deadlock with semaphores
  - Mutual exclusion with monitors
  - Linux Kernel Synchronization

Cooperating Processes

- Once we have multiple processes or threads, it is likely that two or more of them will want to communicate with each other
- Process cooperation (i.e., interprocess communication) deals with three main issues
  - Passing information between processes/threads
  - Making sure that processes/threads do not interfere with each other
  - Ensuring proper sequencing of dependent operations
- These issues apply to both processes and threads
  - Initially we concentrate on shared memory mechanisms
Cooperating Processes

- An *independent* process cannot affect or be affected by the execution of another process.
- A *cooperating* process can affect or be affected by the execution of another process.
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Issues for Cooperating Processes

- Race conditions
  - A *race condition* is a situation where the semantics of an operation on shared memory are affected by the arbitrary timing sequence of collaborating processes.
- Critical regions
  - A *critical region* is a portion of a process that accesses shared memory.
- Mutual exclusion
  - *Mutual exclusion* is a mechanism to enforce that only one process at a time is allowed into a critical region.
Cooperating Processes Approach

- Any approach to process cooperation requires that
  - No two processes may be simultaneously inside their critical regions
  - No assumptions may be made about speeds or the number of CPUs
  - No process running outside of its critical region may block other processes
  - No process should have to wait forever to enter its critical region

Cooperating Processes

Consider a shared, bounded-buffer

```java
public class Buffer {
    private volatile int count = 0;
    private volatile int in = 0, out = 0;
    private Object[] buffer = new Object[10];

    public void enter(Object item) {
        // producer calls this method
    }

    public Object remove() {
        // consumer calls this method
    }
}
```
Cooperating Processes

The remove() operation on the bounded-buffer

```java
// consumer calls this method
public Object remove() {
    Object item;
    while (count == 0)
        ; // do nothing
    // remove an item from the buffer
    --count;
    item = buffer[out];
    out = (out + 1) % buffer.length;
    return item;
}
```

Cooperating Processes

The enter() operation on the bounded-buffer

```java
// producer calls this method
public void enter(Object item) {
    while (count == buffer.length)
        ; // do nothing
    // add an item to the buffer
    ++count;
    buffer[in] = item;
    in = (in + 1) % buffer.length;
}
```
Mutual Exclusion with Busy Waiting

Mutual exclusion via \textit{busy waiting / spin locks}

```
while (TRUE) {
  while (turn != 0) /* loop */;
  critical_region();
  turn = 1;
  noncritical_region();
}
while (TRUE) {
  while (turn != 1) /* loop */;
  critical_region();
  turn = 0;
  noncritical_region();
}
```

- Constantly uses CPU, so it is inefficient
- The above algorithm enforces strict alternation of process execution
- A process may end up being blocked by another process that is not in a critical region

Peterson's solution for mutual exclusion via busy waiting

```
#define FALSE 0
#define TRUE 1
#define N 2 /* number of processes */

int turn; /* whose turn is it? */
int interested[N]; /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
  int other; /* number of the other process */
  other = 1 - process; /* the opposite of process */
  interested[process] = TRUE; /* show that you are interested */
  turn = process; /* set flag */
  while ((turn == process) && interested[other] == TRUE) /* null statement */
  }

void leave_region(int process); /* process: who is leaving */
{
  interested[process] = FALSE; /* indicate departure from critical region */
}
```
Mutual Exclusion with Busy Waiting

Mutual exclusion and busy waiting with hardware support

- Many processors offer a “test and set” atomic instruction
  - An *atomic* instruction cannot be interrupted

```
enter_region:
  TSL REGISTER, LOCK      | copy lock to register and set lock to 1
  CMP REGISTER, #0        | was lock zero?
  JNE enter_region        | if it was non zero, lock was set, so loop
  RET | return to caller; critical region entered

leave_region:
  MOVE LOCK, #0           | store a 0 in lock
  RET | return to caller
```

Mutual Exclusions with Semaphores

- Mutual exclusion using semaphores
  - A *semaphore* is a non-negative integer value
  - Two atomic operations are supported on a semaphore
    - *down* = decrements the value, since the value cannot be negative, the process blocks if the value is zero
    - *up* = increments the value; if there are any processes waiting to perform a down, then they are unblocked
  - Initializing a semaphore to 1 creates a mutual exclusion (*mutex*) semaphore
  - Initializing a semaphore to 0 creates a signaling semaphore
  - Initializing a semaphore to other values can be used for resource allocation and synchronization
Mutual Exclusions with Semaphores

- Mutual exclusion using semaphores
  - Busy waiting versus blocking
    - Semaphores can use busy waiting, but generally they are implemented using blocking
    - With blocking, a process/thread that fails to acquire the semaphore does not loop continuously, instead it is blocked and another thread is scheduled
    - This is more efficient since it does not waste CPU cycles and allows other processes/threads to make progress

Mutual Exclusions with Semaphores

Mutex semaphore implementation

```assembly
mutex_lock:
  TSL REGISTER,MUTEX | copy mutex to register and set mutex to 1
  CMP REGISTER,#0   | was mutex zero?
  JZE ok            | if it was zero, mutex was unlocked, so return
  CALL thread_yield | mutex is busy; schedule another thread
  JMP mutex_lock    | try again later
ok: RET | return to caller; critical region entered

mutex_unlock:
  MOVE MUTEX,#0     | store a 0 in mutex
  RET | return to caller
```
Deadlock with Semaphores

- **Deadlock** is when 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
  
  $P_0 \quad P_1$
  
  $P(S); \quad P(Q);$  
  
  $P(Q); \quad P(S);$  
  
  $:\quad :$  
  
  $V(S); \quad V(Q);$  
  
  $V(Q) \quad V(S);$  
  
- **Starvation** is indefinite blocking when a process is never removed from the semaphore queue in which it was suspended.

Mutual Exclusion with Monitors

- Mutual exclusion using monitors
  - Semaphores are very low-level and error prone
    - The require the programmer to use them properly
  - A **monitor** is a high-level programming language construct that packages data and the procedures to modify the data
    - The data can only be modified using the supplied procedures
    - Only one process is allowed inside the monitor at a time
  - Monitors also have **condition variables** that have two operations, *wait* and *signal*
    - Calling *wait* blocks the calling process
    - Calling *signal* unblocks waiting processes
    - Since condition variables are inside of a monitor there is no race condition when accessing them
Mutual Exclusion with Monitors

```plaintext
monitor example
  integer i;
  condition c;

  procedure producer();
    ...
  end;

  procedure consumer();
    ...
  end;
end monitor;
```

Example of a monitor

Process Cooperation with Messages

- IPC using message passing
  - The previous IPC approaches required some sort of shared memory for communication
    - This does not work well for some sorts of IPC, such as in distributed systems
  - Message passing has different complications, like lost messages
Process Cooperation with Messages

- **IPC using message passing**

```c
#define N 100  /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;
    /* message buffer */
    while (TRUE) {
        item = produce_item();  /* generate something to put in buffer */
        receive(consumer, &m);  /* wait for an empty to arrive */
        build_message(&m, item);  /* construct a message to send */
        send(consumer, &m);  /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;
    for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);  /* get message containing item */
        item = extract_item(&m);  /* extract item from message */
        send(producer, &m);  /* send back empty reply */
        consume_item(item);  /* do something with the item */
    }
}
```

Process Cooperation with Barriers

- **Barrier**
  - Processes perform computation and approach a barrier
  - Processes block when they reach the barrier after finishing their computation
  - Once all processes arrives, then all are unblocked
Cooperation Process Problems

- Classic synchronization problems
  - Bounded-buffer
  - Readers-writers
  - Dining philosophers
  - Sleeping barber
  - We are familiar with all of these from the concurrent programming class

Linux Kernel Synchronization

- Even on a single processor machine, not all kernel requests are handled serially because of interrupts
  - Interrupts cause kernel requests to be interleaved
  - Multiple processors makes matters worse
- Essentially, the kernel handles requests that are generated in one of two ways
  - A process in user mode causes an exception; for example, by executing the `int 0x80` assembly language instruction
  - An external device sends a signal using an IRQ line
- The sequence of instructions executed in kernel mode to handle a kernel request is called a kernel control path
Linux Kernel Control Paths

- In the simplest case, the CPU executes a kernel control path sequentially from the first instruction to the last
- The CPU interleaves kernel control paths when
  - A context switch occurs
  - An interrupt occurs
- Interleaving kernel control paths is necessary to implement preemptive multitasking, but it also improves throughput
- Care must be taken when modifying data structures in kernel control paths to avoid corruption

Linux Synchronization Techniques

- There are four broad synchronization techniques used in Linux
  - Nonpreemptability of processes in Kernel Mode
  - Atomic operations
  - Interrupt disabling
  - Locking
Linux Synchronization Techniques

• Nonpreemptability of processes in Kernel Mode
  – The following assertions always hold in Linux
    • No process running in kernel mode may be replaced by another process, except when it voluntarily releases the CPU
    • Interrupt or exception handlers can interrupt a process in kernel mode, but the CPU must return to the same kernel control path
    • A kernel control path performing interrupt handling can only be interrupted by another interrupt handler
  – This simplifies kernel control paths dealing with nonblocking systems calls, they are atomic
  – Data structures not modified by interrupt handlers can be safely accessed

Linux Synchronization Techniques

• Atomic operations
  – Ensure that an operation is atomic at the CPU level (i.e., executed in a single, non-interruptible instruction)
  – Atomic Intel 80x86 instructions
    • Instructions that make zero or one memory access
    • Read/modify/write instructions such as inc or dec if no other processor has taken the memory bus
    • Read/modify/write instructions prefixed with the lock byte (0xf0) because they lock the memory bus
  – In C you cannot guarantee which instructions the compiler will use, so it is not possible to determine if an operation is atomic
Linux Synchronization Techniques

• Atomic operations
  – Linux provides special functions that are implemented as atomic assembly language instructions, such as
    • `atomic_read(v)`
    • `atomic_set(v, i)`
    • `atomic_add(i, v)`
    • `atomic_sub(i, v)`
    • `atomic_inc(v)`
    • `atomic_dec(v)`
    • `atomic_dec_and_test(v)`
    • ...

• Interrupt disabling
  – Some critical sections are too long to be defined as an atomic operation
  – Disabling interrupts is a mechanism to ensure that a kernel control path is not interrupted
    • The assembly instruction `cli` disables interrupts; `sti` resumes interrupts
    • Page faults can still interrupt the kernel control path
  – Avoid blocking calls with interrupts disabled
  – Linux provides uniprocessor interrupt disabling macros
    • `spin_lock_init()`, `spin_lock()`, `spin_unlock()`, `spin_lock_irq()`, `spin_unlock_irq()`, `write_lock_irqsave()``, `write_unlock_irqrestore()`
    • ...
Linux Synchronization Techniques

• Locking
  – Before entering a critical region, the kernel control path
    acquires a lock for that region
  – There are two locking mechanisms
    • Kernel semaphores
      – Suspends corresponding process if busy
    • Spin locks
      – Kernel control path busy waits
      – Only useful in multiprocessor systems

Linux Synchronization Techniques

• Kernel semaphore
  – Implemented by struct semaphore
    • count field is essentially the semaphore value, but when
      negative it denotes the number of waiting kernel control paths
    • wait field stores the address of a wait queue
    • waking field is also similar to the semaphore value and is
      used to ensure that only one process gets resource (or more
      depending on the initial value of the semaphore)
    – The waking field is protected by disabling interrupts or a spin
      lock and is incremented when an up() is performed and there
      are waiting processes; any waiting processes must acquire the
      waking lock and try to decrement the non-zero waking value
Linux Synchronization Techniques

- Avoid deadlock on semaphores
  - Deadlocks can only occur when a kernel control path requires multiple semaphores to enter a critical region
  - It is rare that a kernel control path needs multiple semaphores
  - In the cases that a kernel control path does need multiple semaphores, they are always acquired in the order of their addresses: lowest address is acquired first