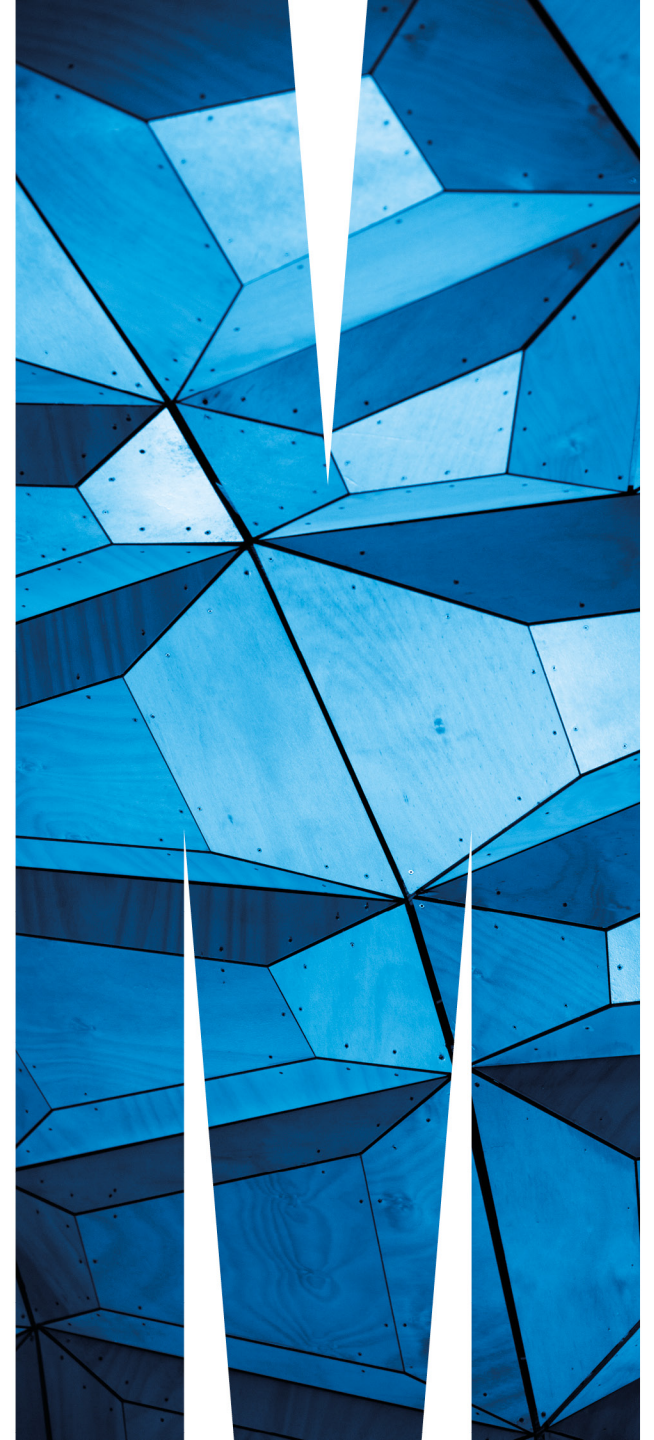


FIT2100 Semester 2 2017

Lecture 7: Interprocess Communication

(Reading: Stallings, Chapter 5 and
Chapter 6)

Jojo Wong



Lecture 7: Learning Outcomes

- Upon the completion of this lecture, you should be able to:
 - Describe how **message passing** can enable **interprocess communication**
 - Understand how **message passing** can be used to enforce **synchronisation**
 - Understand the **readers/writers** problem
 - Discuss the mechanisms for interprocess communication and synchronisation in Unix/Linux systems

Recap: How can processes interact with each other?

Types of Process Interaction

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	<ul style="list-style-type: none"> •Results of one process independent of the action of others •Timing of process may be affected 	<ul style="list-style-type: none"> •Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	<ul style="list-style-type: none"> •Results of one process may depend on information obtained from others •Timing of process may be affected 	<ul style="list-style-type: none"> •Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	<ul style="list-style-type: none"> •Results of one process may depend on information obtained from others •Timing of process may be affected 	<ul style="list-style-type: none"> •Deadlock (consumable resource) •Starvation

Recap: What are the common mechanisms to support concurrency?

Concurrency: Control Mechanisms

Semaphore	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a counting semaphore or a general semaphore .
Binary Semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
Condition Variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures, and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are <i>critical sections</i> . A monitor may have a queue of processes that are waiting to access it.
Event Flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/Messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

What is the purpose of message passing?

Message Passing

- When processes interact with one another, two fundamental requirements must be satisfied:

communication	synchronization
<ul style="list-style-type: none">• to exchange information	<ul style="list-style-type: none">• to enforce mutual exclusion

- **Message passing:**
 - an approach to providing both of these functions
 - Works with distributed systems, shared-memory multiprocessor and and uniprocessor systems

Message Passing

- The actual function is normally provided in the form of a pair of primitive operations:
 - `send(destination,message)`
 - `receive(source,message)`
- A process sends information in the form of a `message` to another process designated by a `destination`
- A process receives information by executing the `receive` primitive, indicating the `source` and the `message`

Message Systems: Design Characteristics

Synchronization

Send
 blocking
 nonblocking
Receive
 blocking
 nonblocking
 test for arrival

Addressing

Direct
 send
 receive
 explicit
 implicit
Indirect
 static
 dynamic
 ownership

Format

Content
Length
 fixed
 variable

Queueing Discipline

FIFO
Priority

Synchronisation: Between Two Processes

Communication of a message between two processes implies synchronisation between two

When a receive primitive is executed in a process there are two possibilities

The receiver cannot receive a message until it has been sent by another process

If a message has previously been sent, the message is received and execution continues

If there is no waiting message, the process is blocked until a message arrives or the process continues to execute, abandoning the receive attempt

Blocking Send and Blocking Receive

- Both sender and receiver are **blocked** until the message is delivered



Sometimes
referred as
'rendezvous'

- Allows for **tight synchronisation** between processes

Non-blocking Send

Non-blocking Send/ Blocking Receive

- Sender continues on but receiver is blocked until the requested message arrives
- Sends one or more messages to a variety of destinations as quickly as possible
- E.g. A server process that provides a service/resource to other processes

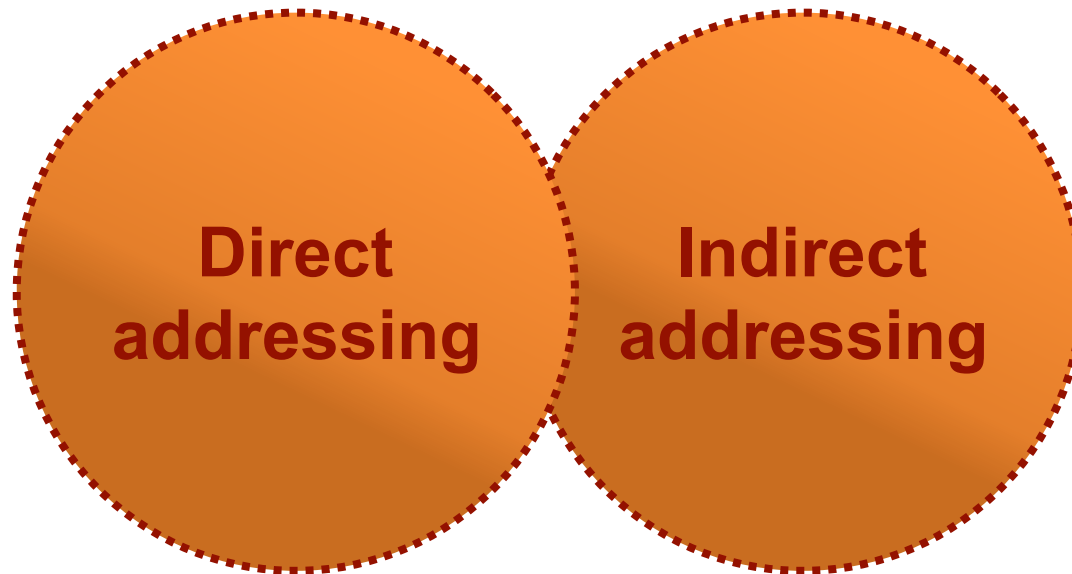
Non-blocking Send/ Non-blocking Receive

- Neither party is required to wait


Non-blocking send and blocking receive are more natural for many concurrent programming tasks

Addressing

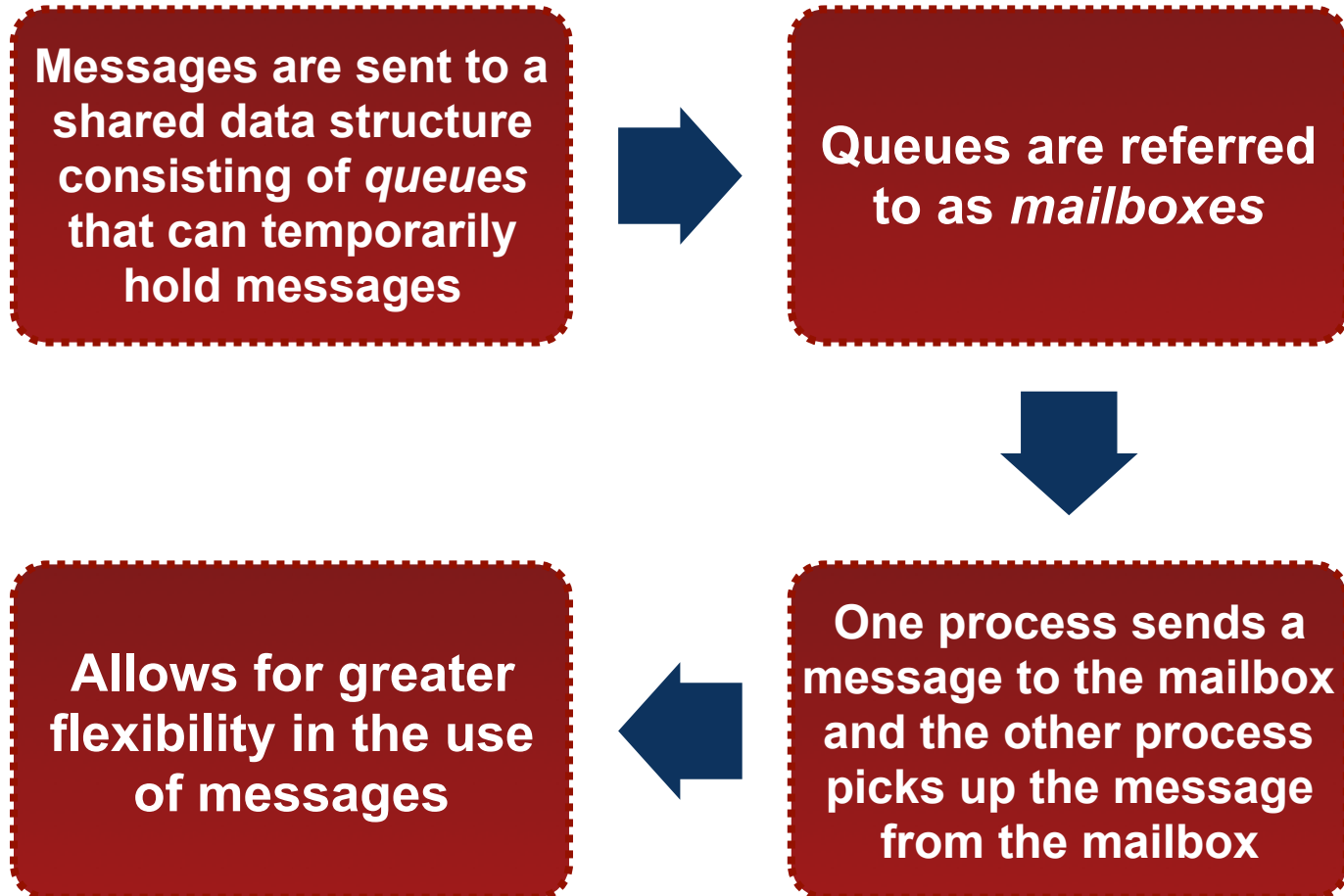
- Schemes for specifying processes in **send** and **receive** primitive operations fall into two categories:



Addressing: Direct

- **send** primitive include a specific *identifier* of the destination process Effective for cooperating concurrent processes
- **receive** primitive can be handled in one of the two ways:
 - Require the process explicitly designate a sending processes
 - *Implicit addressing* — the source parameter of the **receive** primitive possesses a *value* returned when the **receive** operation has been performed

Addressing: Indirect

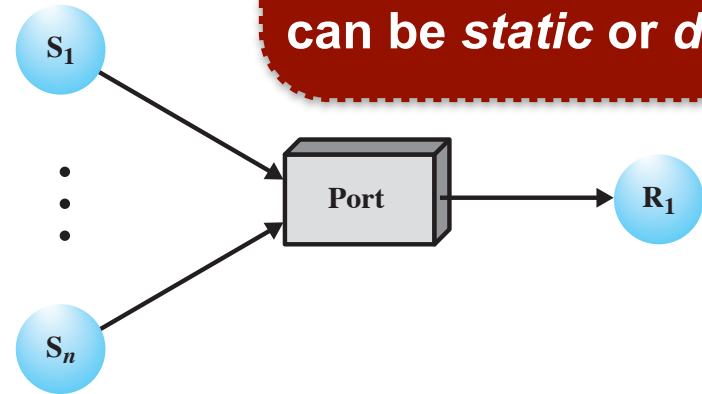


Indirect Process Communication

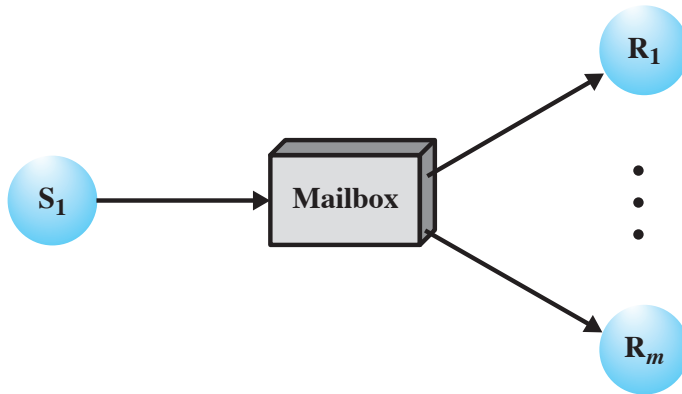
The association of processes to mailboxes can be *static* or *dynamic*



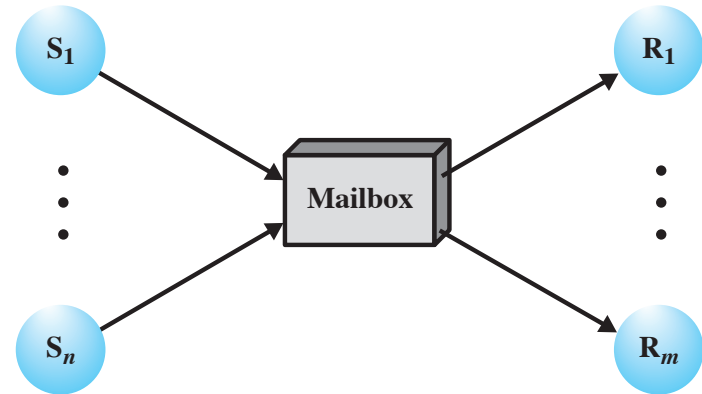
(a) One to one



(b) Many to one



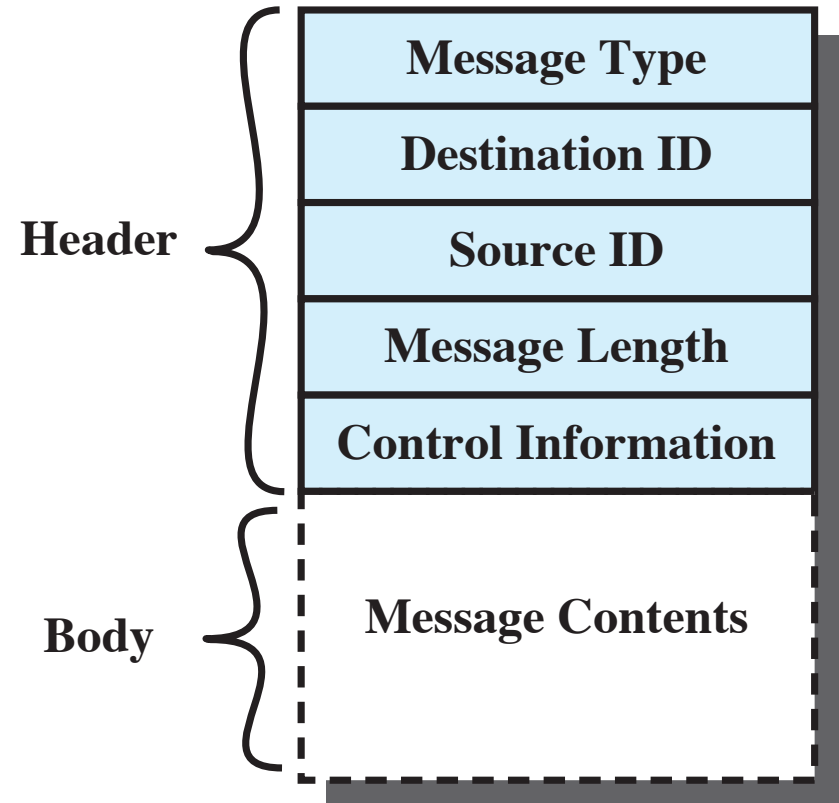
(c) One to many



(d) Many to many

Message Format

- A message is generally divided into two parts:
 - **Header** — contains information about the message (the identifier of the source and intended destination, message length, message type, *control information*)
 - **Body** — contains the actual contents of the message



How can message passing be used
to enforce mutual exclusion?

Mutual Exclusion: Using Messages

```
/* program mutualexclusion */
const int n = /* number of processes */;
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);
        /* critical section */;
        send (box, msg);
        /* remainder */;
    }
}
void main()
{
    create_mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . . , P(n));
}
```

**Blocking
receive and
non-blocking
send**

The Producer/Consumer Problem: Finite Buffer

```
const int
    capacity = /* buffering capacity */ ;
    null = /* empty message */ ;
int i;
void producer()
{
    message pmsg;
    while (true) {
        receive (mayproduce, pmsg);
        pmsg = produce();
        send (mayconsume, pmsg);
    }
}
void consumer()
{
    message cmsg;
    while (true) {
        receive (mayconsume, cmsg);
        consume (cmsg);
        send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++) send (mayproduce,
null);
    parbegin (producer, consumer);
}
```

Using
messages

Two
mailboxes

The Producer/Consumer Problem: Finite Buffer

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Using
semaphores



MONASH
University

What is the readers/writers problem?
(Another classical problem)

The Readers/Writers Problem

A file, a block of main memory, or a collection of processor registers

- A **data area** is *shared* among many processes
 - Some processes **only read** the data area (readers)
 - Some **only write** to the data area (writers)
- **Conditions** that must be satisfied:
 1. Any number of readers may *simultaneously* read the file.
 2. Only one writer at a time may write to the file.
 3. If a writer is writing to the file, no reader may read it.

The Readers/Writers Problem: Using Semaphores

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
void writer()
{
    while (true) {
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
    }
}
void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

Readers have
priority

Only the first
reader should
wait on the
semaphore *wsem*

The Readers/Writers Problem: Using Semaphores

```
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        semSignal (rsem);
        semSignal (z);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
void writer ()
{
    while (true) {
        semWait (y);
        writecount++;
        if (writecount == 1) semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount--;
        if (writecount == 0) semSignal (rsem);
        semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Semaphore *rsem* inhibits all the readers if at least one writer desiring access to the data area

Writers have priority

The Readers/Writers Problem: State of Process Queues

Readers only in the system	<ul style="list-style-type: none">• <i>wsem</i> set• no queues
Writers only in the system	<ul style="list-style-type: none">• <i>wsem</i> and <i>rsem</i> set• writers queue on <i>wsem</i>
Both readers and writers with read first	<ul style="list-style-type: none">• <i>wsem</i> set by reader• <i>rsem</i> set by writer• all writers queue on <i>wsem</i>• one reader queues on <i>rsem</i>• other readers queue on <i>z</i>
Both readers and writers with write first	<ul style="list-style-type: none">• <i>wsem</i> set by writer• <i>rsem</i> set by writer• writers queue on <i>wsem</i>• one reader queues on <i>rsem</i>• other readers queue on <i>z</i>

The Readers/Writers Problem: Using Messages

**Controller has
access to the
shared data area**

```
void reader(int i)
{
    message rmsg;
    while (true) {
        rmsg = i;
        send (readrequest, rmsg);
        receive (mbox[i], rmsg);
        READUNIT ();
        rmsg = i;
        send (finished, rmsg);
    }
}

void writer(int j)
{
    message rmsg;
    while(true) {
        rmsg = j;
        send (writerequest, rmsg);
        receive (mbox[j], rmsg);
        WRITEUNIT ();
        rmsg = j;
        send (finished, rmsg);
    }
}
```

```
void controller()
{
    while (true)
    {
        if (count > 0) {
            if (!empty (finished)) {
                receive (finished, msg);
                count++;
            }
            else if (!empty (writerequest)) {
                receive (writerequest, msg);
                writer_id = msg.id;
                count = count - 100;
            }
            else if (!empty (readrequest)) {
                receive (readrequest, msg);
                count--;
                send (msg.id, "OK");
            }
        }
        if (count == 0) {
            send (writer id, "OK");
            receive (finished, msg);
            count = 100;
        }
        while (count < 0) {
            receive (finished, msg);
            count++;
        }
    }
}
```

What are the control mechanisms in
Unix?

Unix: Concurrency Control Mechanisms

- Unix provides a variety of mechanisms for interprocess communication and synchronisation:

Pipes

Messages

**Shared
Memory**

Semaphores

Signals

Unix: Pipes

- *Circular buffers* allowing two processes to communicate on the producer-consumer model
 - **First-in-first-out queue** — written by one process and read by another
- **Mutual exclusion** is enforced
 - Only one process can access a pipe at a time
- Two types of pipes:
 - **Named** — can be shared by related or unrelated processes
 - **Unnamed** — can only be shared by related processes

Unix: Messages

As a selection
criterion by the
receiver

- A block of bytes with an accompanying type
- UNIX provides **msgsnd** and **msgrcv** system calls for processes to engage in message passing
- Associated with each process is a **message queue**, which functions like a mailbox

A process will be
blocked when attempts
to send a message to a
full queue

Unix: Shared Memory

- Fastest form of interprocess communication
- Common block of (virtual) memory shared by multiple processes
- Permission is read-only or read-write for a process
- Mutual exclusion constraints are *not* part of the shared-memory facility — but must be provided by the processes using the shared memory

On a per-process basis

Unix: Semaphores

- Generalisation of **semWait** and **semSignal** primitive operations
- Associated with the semaphore are *queues* of blocked process on that semaphore
- Semaphores are created in *sets* — a semaphore set consists of one or more semaphores

A semaphore consists of the following elements:

- current value of the semaphore
- process ID of the last process to operate on the semaphore
- number of processes waiting for the semaphore value to be greater than its current value
- number of processes waiting for the semaphore value to be zero

Unix: Signals

- A software mechanism that informs a process of the occurrence of asynchronous events
- All signals are treated equally — no priority imposed
- A signal is delivered by updating a field (bit) in the process table for the process to which the signal is being sent
- A process may respond to a signal by:
 - Performing some default action (e.g. termination)
 - Executing a signal-handler function
 - Ignoring the signal

Unix: Signals

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

What are the additional mechanisms
in Linux?

Linux: Kernel Concurrency Mechanisms

- Mechanisms **used within the kernel** to provide concurrency the execution of kernel code

**Atomic
Operations**

Spinlocks

Semaphores

**Barrier
Operations**

Linux: Atomic Operations

To avoid
simple race
conditions

- Simplest of the approaches to kernel synchronisation
- Atomic operations **execute without interruption and without interference**

Integer Operations

operate on an
integer variable

typically used to
implement
counters

Bitmap Operations

operate on one of
a sequence of bits
at an arbitrary
memory location
indicated by a
pointer variable

Linux: Atomic Operations

Atomic Integer Operations	
ATOMIC_INIT (int i)	At declaration: initialize an atomic_t to i
int atomic_read(atomic_t *v)	Read integer value of v
void atomic_set(atomic_t *v, int i)	Set the value of v to integer i
void atomic_add(int i, atomic_t *v)	Add i to v
void atomic_sub(int i, atomic_t *v)	Subtract i from v
void atomic_inc(atomic_t *v)	Add 1 to v
void atomic_dec(atomic_t *v)	Subtract 1 from v
int atomic_sub_and_test(int i, atomic_t *v)	Subtract i from v; return 1 if the result is zero; return 0 otherwise
int atomic_add_negative(int i, atomic_t *v)	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
int atomic_dec_and_test(atomic_t *v)	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
int atomic_inc_and_test(atomic_t *v)	Add 1 to v; return 1 if the result is zero; return 0 otherwise
Atomic Bitmap Operations	
void set_bit(int nr, void *addr)	Set bit nr in the bitmap pointed to by addr
void clear_bit(int nr, void *addr)	Clear bit nr in the bitmap pointed to by addr
void change_bit(int nr, void *addr)	Invert bit nr in the bitmap pointed to by addr
int test_and_set_bit(int nr, void *addr)	Set bit nr in the bitmap pointed to by addr; return the old bit value
int test_and_clear_bit(int nr, void *addr)	Clear bit nr in the bitmap pointed to by addr; return the old bit value
int test_and_change_bit(int nr, void *addr)	Invert bit nr in the bitmap pointed to by addr; return the old bit value
int test_bit(int nr, void *addr)	Return the value of bit nr in the bitmap pointed to by addr

Linux: Spinlocks

- Most common technique for protecting a critical section in Linux
- Can only be **acquired by one thread at a time**
 - Any other thread will keep trying (spinning) until it can acquire the lock
- Built on **an integer location in memory** that is checked by each thread before it enters its critical section
- Effective in situations where the wait time for acquiring a lock is expected to be very short
- **Disadvantage:**
 - Locked-out threads continue to execute in a *busy-waiting* mode

Linux: Spinlocks

<code>void spin_lock(spinlock_t *lock)</code>	Acquires the specified lock, spinning if needed until it is available
<code>void spin_lock_irq(spinlock_t *lock)</code>	Like <code>spin_lock</code> , but also disables interrupts on the local processor
<code>void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)</code>	Like <code>spin_lock_irq</code> , but also saves the current interrupt state in flags
<code>void spin_lock_bh(spinlock_t *lock)</code>	Like <code>spin_lock</code> , but also disables the execution of all bottom halves
<code>void spin_unlock(spinlock_t *lock)</code>	Releases given lock
<code>void spin_unlock_irq(spinlock_t *lock)</code>	Releases given lock and enables local interrupts
<code>void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)</code>	Releases given lock and restores local interrupts to given previous state
<code>void spin_unlock_bh(spinlock_t *lock)</code>	Releases given lock and enables bottom halves
<code>void spin_lock_init(spinlock_t *lock)</code>	Initializes given spinlock
<code>int spin_trylock(spinlock_t *lock)</code>	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
<code>int spin_is_locked(spinlock_t *lock)</code>	Returns nonzero if lock is currently held and zero otherwise

Linux: Semaphores

- At the **user level**:
 - Same as those semaphores in Unix
- At the **kernel level**:
 - Implemented as functions within the kernel
- Three types of kernel semaphores:
 - Binary semaphores
 - Counting semaphores
 - Reader-writer semaphores



MUTEX

Linux: Semaphores

Traditional Semaphores	
<code>void sema_init(struct semaphore *sem, int count)</code>	Initializes the dynamically created semaphore to the given count
<code>void init_MUTEX(struct semaphore *sem)</code>	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
<code>void init_MUTEX_LOCKED(struct semaphore *sem)</code>	Initializes the dynamically created semaphore with a count of 0 (initially locked)
<code>void down(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
<code>int down_interruptible(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns <code>-EINTR</code> value if a signal other than the result of an up operation is received
<code>int down_trylock(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
<code>void up(struct semaphore *sem)</code>	Releases the given semaphore
Reader-Writer Semaphores	
<code>void init_rwsem(struct rw_semaphore, *rwsem)</code>	Initializes the dynamically created semaphore with a count of 1
<code>void down_read(struct rw_semaphore, *rwsem)</code>	Down operation for readers
<code>void up_read(struct rw_semaphore, *rwsem)</code>	Up operation for readers
<code>void down_write(struct rw_semaphore, *rwsem)</code>	Down operation for writers
<code>void up_write(struct rw_semaphore, *rwsem)</code>	Up operation for writers

Linux: Barrier Operations

- It is important that **reads or writes are executed in the order specified**
 - E.g. use of information made by another thread or hardware device
- **Memory barriers:**
 - Enforce the order in which program instructions are executed

**machine instructions (not
high-level programming
instructions)**

Linux: Barrier Operations

<code>rmb()</code>	Prevents loads from being reordered across the barrier
<code>wmb()</code>	Prevents stores from being reordered across the barrier
<code>mb()</code>	Prevents loads and stores from being reordered across the barrier
<code>Barrier()</code>	Prevents the compiler from reordering loads or stores across the barrier
<code>smp_rmb()</code>	On SMP, provides a <code>rmb()</code> and on UP provides a <code>barrier()</code>
<code>smp_wmb()</code>	On SMP, provides a <code>wmb()</code> and on UP provides a <code>barrier()</code>
<code>smp_mb()</code>	On SMP, provides a <code>mb()</code> and on UP provides a <code>barrier()</code>

**SMP = Symmetric
multiprocessor;
UP = uniprocessor**

Summary of Lecture 7

- Two common approaches to support mutual exclusion for interprocess communication:
 - **Semaphores** — for signal among processes and can be readily used to enforce mutual exclusion
 - **Messages** — an effective means of interprocess communication and useful for the enforcement of mutual exclusion

Next week: Memory management