Shared Memory Programming: Threads, Semaphores, and Monitors

CPS343

Parallel and High Performance Computing

Spring 2020

Outline

1. Processes
   - The Notion and Importance of Processes
   - Process Creation and Termination
   - Inter-process Communication and Coordination
2. Threads
   - Need for Light Weight Processes
   - Differences between Threads and Processes
   - Implementing Threads
3. Mutual Exclusion and Semaphores
   - Critical Sections
   - Monitors

Outline

1. Processes
   - The Notion and Importance of Processes
   - Process Creation and Termination
   - Inter-process Communication and Coordination
2. Threads
   - Need for Light Weight Processes
   - Differences between Threads and Processes
   - Implementing Threads
3. Mutual Exclusion and Semaphores
   - Critical Sections
   - Monitors

What is a process?

A process is a **program in execution**.

At any given time, the status of a process includes:

- The **code** that it is executing (e.g. its **text**).
- Its **data**:
  - The values of its static variables.
  - The contents of its stack – which contains its local variables and procedure call/return history.
- The contents of the various **CPU registers** – particularly the program counter, which indicates what instruction the process is to execute next.
- Its **state** – is it currently able to continue execution, or must further execution wait until some event occurs (e.g. the completion of an IO operation it has requested)?

What is a process?

The chief task of an operating system is to manage a set of processes.

Single Core CPU:

- On system with one single-core CPU, there is, at any given time, at most only one **current process** – namely the one whose code the hardware PC register is actually pointing to.
- Given that there may be many processes that wish to use the CPU, the operating system kernel is responsible for **scheduling** the processes – i.e. determining which process should be current at any given time.

Multiple CPU/Multiple Core:

- Of course, on a system with multiple CPU cores there can be more than one current process - but at most one per core.

The process control block

To manage the various processes, the operating system maintains a set of data structures called **process control blocks (PCB's)** – one per process.

Each PCB contains information about its process:

- The **state** of the process.
- The **values of the CPU registers** (relevant only when the process is not running; when it is running, these values change constantly).
  Note: The set of values may include both values visible to the programmer and hidden registers such as memory base and limit registers.
- Memory management information
- Scheduling, accounting, and resource allocation information.

Context Switches

One major use of the PCB's is in conjunction with a **context switch** operation.

- A context switch occurs whenever the a new process is given use of the CPU in place of the one that is currently running.
- During a context switch, the first thing that must happen is that the register values associated with the currently running process must be stored into its PCB, to be saved until the process next gets a chance to run.
- Then, the stored register values found in the PCB of the new process are copied into the CPU registers.

Outline

1. Processes
   - The Notion and Importance of Processes
   - Process Creation and Termination
   - Inter-process Communication and Coordination
2. Threads
   - Need for Light Weight Processes
   - Differences between Threads and Processes
   - Implementing Threads
3. Mutual Exclusion and Semaphores
   - Critical Sections
   - Monitors

Parents and Children

The creator process is called the **parent** and the new process is called a **child**. (By repeated use of the create process service, a parent could create multiple children, of course.)

Various aspects of the relationship between a parent and child are handled differently by different systems. These include the **memory image**, **resources**, and **process lifetime**.

Process Resources

Memory Image On a Linux/Unix system, a new child process is created using the `fork()` system call. The new process shares its parent's text and inherits a copy of its parents data. The child can later acquire its own, separate text by using the `execve()` system call.

Resources Systems impose upper limits on how large a quantity of various system resources a process can claim (e.g. memory, open files etc.) On some systems, the resources used by a child are counted against its parents quota; on others each child has its own quota.

Process Lifetime Possibility of the child living on after the parent terminates. On some systems termination of a parent process automatically causes its children to terminate as well.

Process Termination

A process may terminate itself by invoking a suitable system service.

On many systems, a parent process may also terminate one of its children through a system service call.

The operating system may terminate processes for various reasons (e.g., deadlock recovery).

Outline

1. Processes
   - The Notion and Importance of Processes
   - Process Creation and Termination
   - Inter-process Communication and Coordination
2. Threads
   - Need for Light Weight Processes
   - Differences between Threads and Processes
   - Implementing Threads
3. Mutual Exclusion and Semaphores
   - Critical Sections
   - Monitors

Shared Memory

- Some systems allow processes to **share physical memory**, so that data written to a given location by one process can be read by another.
- In this case the operating system must provide services for declaring a given region of memory to be shared and for attaching to a shared region that was created by another process.
- Some architectures supporting NUMA (non-uniform memory access) pool all memory on all nodes into one large, shared memory resource.

Message Passing

- Most systems allow processes to communicate with one another by means of **messages**.
- The operating system must provide facilities for sending and receiving messages.
- These may be transmitted within a single computer, or over a network.

Processes Synchronization

- Most systems provide some services for **synchronization of processes**, so that one process can wait until another process has reached a certain point in its execution.
- This is especially important when shared memory is used – a process that intends to access data that another has written has to wait until the other process has done so.

Outline

1. Processes
   - The Notion and Importance of Processes
   - Process Creation and Termination
   - Inter-process Communication and Coordination
2. Threads
   - Need for Light Weight Processes
   - Differences between Threads and Processes
   - Implementing Threads
3. Mutual Exclusion and Semaphores
   - Critical Sections
   - Monitors

Processes and Threads

Around 1970 the notion of **process** was been broadened to include two kinds of processes – traditional or **heavy weight** processes, and **light weight** processes – also known as **threads**.

The motivation for this development was a desire to balance considerations of efficiency with considerations of software modularity.

The Case for Light Weight Processes

- The use of multiple processes (of any sort) entails periodic *context switches*, in which the CPU is passed from one process to another.
- Unfortunately, a context switch operation can be quite time-consuming, especially if a significant number of memory-management related values must be saved and restored.
- Thus, one consideration in designing an efficient system is avoiding excess context switching.

The Case for Light Weight Processes

- The trend in the design of complex software systems, however, was to organize these systems as a set of cooperating processes, rather than as a single monolithic process.
- For example, a **producer/consumer** sort of problem is often structured as a pair of processes, perhaps with a third buffer process mediating between them.
- Another example: server systems (such as web servers) often use one process per client served.

The Case for Light Weight Processes

- The structuring of tasks as a collection of concurrent processes leads, of course, to increased frequency of context switching - which conflicts with the goal of minimizing context switches for efficiency reasons.
- Further, if a task is structured as a collection of cooperating processes, then message-passing overhead is involved when the processes need to communicate with each other. (This is not an issue when a task is structured as a single process, or when a collection of processes operate independently of one another.)

Outline

1 Processes
- The Notion and Importance of Processes
- Process Creation and Termination
- Inter-process Communication and Coordination

2 Threads
- Need for Light Weight Processes
- Differences between Threads and Processes
- Implementing Threads

3 Mutual Exclusion and Semaphores
- Critical Sections
- Monitors

Threads

One way to resolve the tension between modularity and efficiency is to allow a single process to be structured as a set of individual **threads**.

- *All the threads share the same memory context* (code and data). There is no memory protection between threads in the same process – any thread may alter any data item.
- *Each thread has its own register context* (including PC and SP), *and its own stack.*

Threads

Threads have two major advantages over processes:

1. A context switch between threads it the same process is much faster than a context switch between processes, since only register context needs to be saved and restored - not memory management context. Thus one can have the modularity advantages of separate processes with less overhead.
2. Since all the threads share the same data, information sharing between the threads does not require the overhead of message passing.

Outline

1. Processes
   - The Notion and Importance of Processes
   - Process Creation and Termination
   - Inter-process Communication and Coordination

2. Threads
   - Need for Light Weight Processes
   - Differences between Threads and Processes
   - Implementing Threads

3. Mutual Exclusion and Semaphores
   - Critical Sections
   - Monitors

Thread Implementation

Threads can be implemented in two different ways:

1. By the use of library routines (e.g. Pthreads), independent of the operating system. (I.e. the operating system manages a single process; the process manages its own threads.)
2. By adding a threads facility to the operating system. System services are required for:
   - Creating a new process (initially consisting of a single thread).
   - Creating a new thread within an existing process.

Thread Implementation

- The first approach is simpler, and minimizes the overhead involved in a context switch between threads (since no system call is involved).
- The second approach, however, allows one thread in a process to block itself on an IO operation while allowing the other threads to continue executing. For this reason, some systems make both approaches available.

Thread Implementation

- On a system that provides threads, processes normally do not share memory, but threads within a given process always share memory.
- Processes usually communicate among themselves by message passing.
- Threads normally communicate through shared memory (though they may also use message passing among themselves if desired.)

Outline

1. Processes
   - The Notion and Importance of Processes
   - Process Creation and Termination
   - Inter-process Communication and Coordination
2. Threads
   - Need for Light Weight Processes
   - Differences between Threads and Processes
   - Implementing Threads
3. Mutual Exclusion and Semaphores
   - Critical Sections
   - Monitors

Race Conditions

Consider a multithreaded *master-worker* program where the *master* maintains a list of tickets corresponding to jobs that must be completed.

The following steps are run in parallel until all work has been completed:

- each *worker* (1) requests a ticket from them master and then (2) does the corresponding work.
- the *master* (1) receives a request from a *worker*, (2) provides the current ticket, and (3) increments the ticket.

Race Conditions

Now suppose we have the following sequence of events, given in chronological order:

worker A requests a ticket	
	the *master* receives a request for a ticket
	the *master* gives the current ticket to *worker A*
worker A starts working	
a *worker B* requests a ticket	
	the *master* gives the current ticket to *worker B*
worker B starts working	
	the *master* increments the ticket
	the *master* increments the ticket

Race Conditions

We have a problem here – because the master's operations on the ticket were not **atomic**, two different worker threads were given the same ticket.

This example may seem contrived because it depends on a very specific (and perhaps improbable) sequence of events. *But it is the very rarity of these situations that makes them so insidious!* They will probably arise in practice on rare occasions, but are very difficult to locate and fix because they cannot be reproduced at will.

These situations are called **race conditions**. A race condition occurs whenever the outcome of the execution depends on the particular order that instructions are executed.

Critical Section

We've just seen a simple illustration of the **critical section problem**: When two or more processes share data in common, the code that updates this data is called a *critical section.* To ensure integrity of the shared data, we require that at most one process is executing in its critical section for a given data structure at one time.

Critical Section

To deal with critical sections, we need a methodology that guarantees:

- **Mutual exclusion**: Under no circumstances can two processes be in their critical sections (for the same data structure) at the same time.
- **Progress**: At no time do we have a situation where a process is forced to wait forever for an event that will never occur. (This is also known as the *no deadlock* requirement.)
- **No starvation**: No process waiting for a given resource can be forced to wait forever while other processes repeatedly lock and unlock the same resource. (This is also called the *bounded wait* or the *fairness* requirement.)

Semaphores

One approach to dealing with the critical section problem is the use of a facility first proposed by Dijkstra called a **semaphore**.

- In its simplest form, a semaphore is a boolean variable with two indivisible hardware operations possible on it:

$P(s)$	*wait(s)*	`while (s==0); s=0;`
$V(s)$	*signal(s)*	`s=1;`

- The names P and V are from two Dutch words *proberen* (to wait) and *verhogen* (to increment).
- This is a **binary semaphore**.

Counting Semaphores

In practice, we often use one of several possible generalizations of the basic semaphore.

One such generalization is called a **counting semaphore**, which can assume any (non-negative) integer value. The operations, which must be done indivisibly, are

$P(s)$	*wait(s)*	`while (s<=0); s=s-1;`
$V(s)$	*signal(s)*	`s=s+1;`

Note that the binary semaphore is a special case, with s constrained to assume only the values 0,1.

Semaphore Example: Bounded Buffer Problem

Assume that there are n buffers, each capable of holding a single item. We use three semaphores: `empty_slot` and `full_slot` to count the empty and full buffers and `mutex` to provide mutual exclusion for operations on the buffer pool. Assume `mutex` is initialized to 1, `empty_slot` is initialized to n and `full_slot` is initialized to 0.

Producer Process

```
while ( true ) {
  . . .
  produce an item
  . . .
  wait(empty_slot);
  wait(mutex);
  . . .
  insert item to buffer
  . . .
  signal(mutex);
  signal(full_slot);
}
```

Consumer Process

```
while ( true ) {
  wait(full_slot);
  wait(mutex);
  . . .
  remove item from buffer
  . . .
  signal(mutex);
  signal(empty_slot);
  . . .
  consume the item
  . . .
}
```

Semaphores can be tricky...

There are several challenges in using semaphores to address the critical section problem:

1. The burden is on the programmer to use the semaphore. There is no way to stop a programmer from updating a shared variable without first doing the necessary *wait()*, short of manually inspecting all code. Thus, mutual exclusion can be lost through carelessness or laziness.
2. Subtle errors are easily made, and can lead to big problems. E.g. suppose a particular critical region needs access to data protected by two different semaphores S_1 and S_2. Then the code

Thread 0

```
wait(S1); wait(S2);
<critical region>
signal(S1); signal(S2)
```

Thread 1

```
wait(S2); wait(S1);
<critical region>
signal(S2); signal(S1)
```

could lead to deadlock if the two threads both did their first *wait()* before either did its second.

Outline

1. Processes
   - The Notion and Importance of Processes
   - Process Creation and Termination
   - Inter-process Communication and Coordination
2. Threads
   - Need for Light Weight Processes
   - Differences between Threads and Processes
   - Implementing Threads
3. Mutual Exclusion and Semaphores
   - Critical Sections
   - Monitors

Monitors

A **monitor** provides synchronization between threads through *mutual exclusion* and *condition variables*.

- A *thread-safe* class is an example of a monitor.
- A *mutex* is mechanism that provides mutual exclusion, often through an underlying binary semaphore.
- A *condition variable* is a container for a set of threads waiting for some *condition* to be met.

Condition Variables

Condition variables provide operations *wait* and *signal* similar to those in a semaphore, but there are some important differences:

- A process executing a *wait* is always blocked.
- A *signal* executed when the condition's queue is empty has no effect; it is not remembered.
- The queue of threads waiting on a given signal is usually FIFO.

To emphasize this difference, in many monitor implementations *notify* is used rather than *signal*.

Example Monitor: Bounded Buffer (1)

```
monitor Buffer
{
  public:
    Buffer( void );
    void insert( char c );
    char remove( void );
  private:
    const int size = 100;
    char      buff[size];
    int       in, out;
    bool      full;
    condition not_empty, not_full;
};

Buffer::Buffer( void )
{
    in = 0;
    out = 0;
    full = false;
};
```

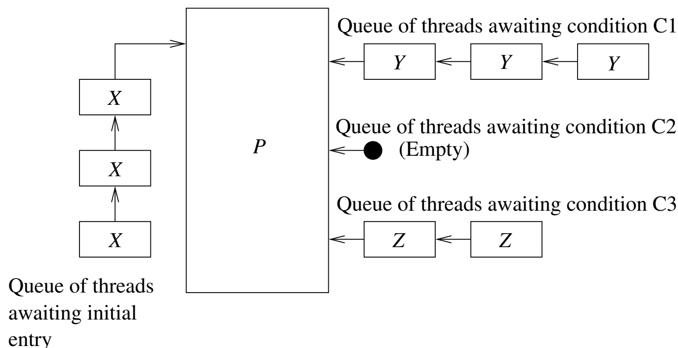
Example Monitor: Bounded Buffer (2)

```
void Buffer::insert( char c )
{
    if ( full ) wait( not_full );
    buff[in] = c;
    in = ( in + 1 ) % size;
    full = ( in == out );
    notify( not_empty );
};

char Buffer::remove( void )
{
    int c;
    if ( !full && in == out ) wait( not_empty );
    c = buff[out];
    out = ( out + 1 ) % size;
    full := false;
    notify( not_full );
    return c;
};
```

A Simple Monitor

In the simplest sort of monitor, we impose the restriction that a *signal* operation, if it appears in a given procedure, must appear at the very end. When a *signal* is executed, the calling thread leaves the monitor, and the first thread on the queue awaiting that condition is admitted. (This is a Lambson-Redell monitor). We can picture such a monitor as follows:



A Simple Monitor

P is the thread that is currently in the monitor. When P leaves, one other thread will be admitted.

- If P exits by signalling condition $C1$, the first Y thread will be admitted.
- If P exits by signalling condition $C3$, the first Z thread will be admitted.
- If P exits by signalling condition C2, or simply exits without signalling, the first X thread will be admitted.

A More Complex Monitor

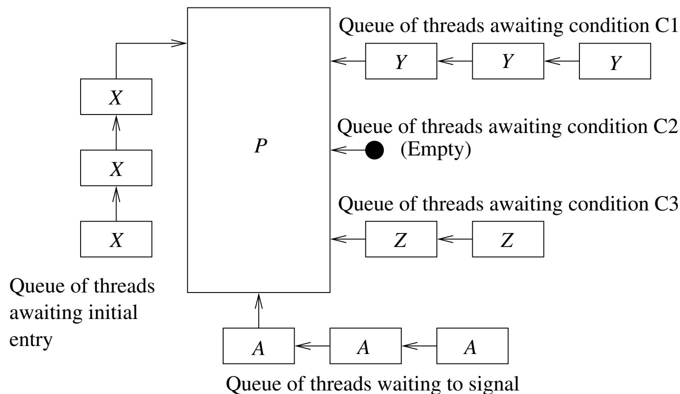
As a further extension, we may remove the restriction that a thread can only signal as its last act. (This would also allow a given call to generate more than one signal.) This is essentially the original monitor as suggested by *Hoare*.

This raises a problem — when a signal awakes a thread, and the signaler wishes to remain in the monitor, one must yield to the other.

One way to handle this is as follows: if a procedure contains a signal other than as its last statement, the calling thread is suspended while the awakened thread completes its work. The calling thread has priority to re-enter the monitor over any processes awaiting at the main gate.

A More Complex Monitor

This can be pictured as follows:



A More Complex Monitor

P is currently in the monitor. When P leaves, one other thread will be admitted.

- If P exits by signalling condition $C1$, the first Y thread will be admitted.
- If P exits by signalling condition $C3$, the first Z thread will be admitted.
- If P exits by signalling condition $C2$, or simply exits without signalling, the A process will be re-admitted.

If P signals with work yet to be done, the above will hold, but P will be added to the queue of waiting signalers (and become an A thread). Note that this queue could grow to any size, if each awakened process in turn awakens another in mid-stream.