Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 3:

- Processes and Threads
- Synchronization Primitives in Shared Memory Programming

Processes and Programs

A ***process*** is an instance of an executing program.

A ***program*** is a binary file that contains a range of information that describes how to construct a process at runtime.

- Binary Format Identification
- Machine Code
- Program Entry-Point Address
- Data
- Symbol & Relocation Tables
- Shared Library & Dynamic Library Information
- Other Information

Processes

Definition:

A process is an abstract entity, defined by the kernel of an OS, to which system resources are allocated in order to execute a program.

From the kernel's point of view, a process consists of:

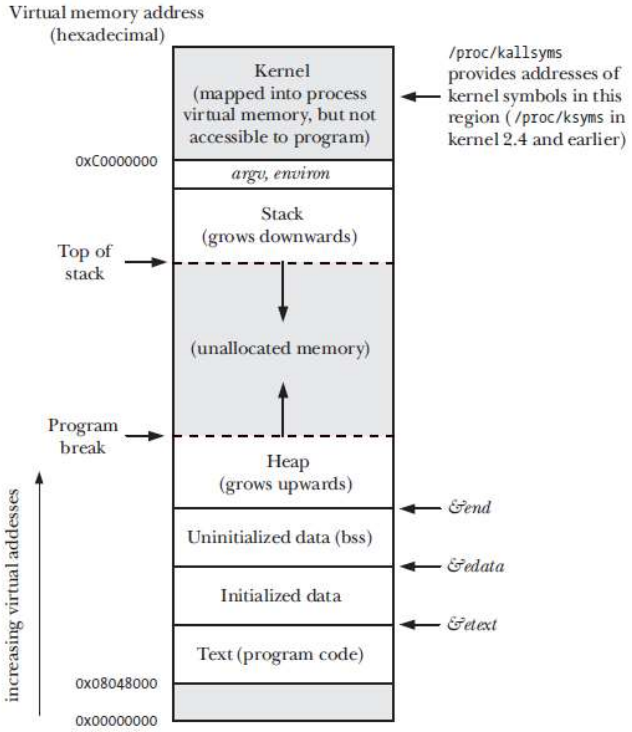
- User-space memory containing both program code and data used by the code
- A range of kernel data structures for maintaining information about the state of the process such as
  - page table
  - open file descriptors
  - signal delivery & handling

Memory Layout

The memory allocated to each process is divided into a number of parts, usually referred to as ***segments***:

- The ***text segment*** contains the machine code of the program run by the process.
  - The text segment is made read-only so that the process does not accidentally modify its own instructions.
  - Many processes may be running the same program so the text segment is made sharable so that a single program can be mapped onto the virtual address space of all the processes.
- The ***initialized data segment*** contains global and static variables that are explicitly initialized.
- The ***uninitialized data segment*** contains global and static variables that are not initialized.
  - Before starting the program, the loader initializes all memory in this segment to 0.
- The ***stack segment*** is a dynamically growing and shrinking segment containing stack frames.
  - One stack frame is allocated for each currently called function.
- The ***heap segment*** is an area from which memory can be dynamically allocated or deallocated at runtime.

Memory Layout



Typical memory layout of a process on Linux/x86-32

Virtual Memory Management

The memory management subsystem divides the memory used by each process into small fixed-size units called ***pages***.

- On typical OSes/architectures, the size of a page is 4 KB.
- Modern OSes/architectures also support larger page sizes called ***Huge Pages***, e.g. 2 MB, 1GB etc.

Correspondingly, the physical main memory (RAM) is divided into ***page frames*** of the same size.

- At any one time, only a subset of the pages of a process need to be present in main memory: these pages form the so-called ***resident set***.
- Copies of the unused pages are stored in the ***swap area***, a reserved area of disk space and loaded into physical memory only as required.
- When a process requests a memory reference belonging to a page that is not currently in the resident set, a ***page fault*** occurs, where the kernel suspends the execution of the process while the page is loaded from disk into main memory.

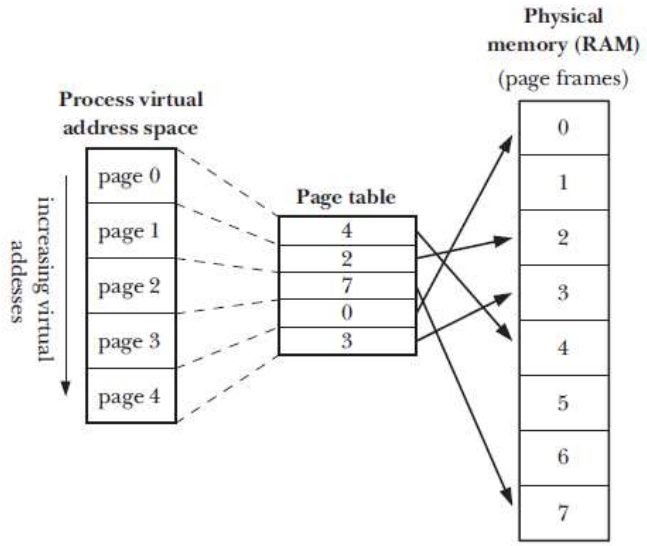
Virtual Memory Management

To support this organization, the OS maintains a ***page table*** for each process.

- The page table describes the location of each page in the process's ***virtual address space***.
- Each entry in the page table either indicates the location of a virtual page in RAM or indicates that it is currently resident on disk.
- Not all address ranges in the process's virtual address space require page-table entries.
- If a process attempts to reference an address for which there is no corresponding page-table entry, it receives a ***fault***: On Linux, a ***SIGSEGV*** signal is delivered.
  - This concept is used to implement the ***null pointer exception*** in high-level programming languages.

Virtual Memory Management

The memory management subsystem separates the ***virtual address space*** of a process from the ***physical address space*** of RAM.



Overview of Virtual Memory

Virtual Address Space

This organization provides the following ***advantages***:

- Processes are isolated from each other so that one process cannot read or modify the memory of another process.
  - This is achieved by having the page-table entries point to distinct sets of pages in RAM or in the swap area.
- Where appropriate, two or more processes can share memory.
  - The OS achieves ***shared memory*** by having page-table entries in different processes refer to the same pages of RAM.
- Programmers, and tools such the compiler and linker do not concern themselves with the physical layout of computer programs in RAM.
- Only a part of a program needs to reside in RAM so the program loads and runs faster.
  - The memory footprint of a process can exceed the amount of RAM available
- Since each process requires less RAM at any one time, more processes can simultaneously be held in RAM and this leads to better CPU utilization since there is an increased likelihood that, at any moment, there are more processes resident in RAM that can be readily scheduled on the CPU.

Shared Memory

The sharing of memory between processes occur in **two common circumstances**:

- Multiple processes that execute the same executable can share a single copy of the machine code, i.e., the ***text segment***.
  - This type of sharing is implicitly performed by the ***loader*** and the ***dynamic linker***.
- Processes use some ***system calls*** to explicitly request sharing of memory regions with other processes for the purpose of ***interprocess communication***, commonly referred to as ***IPC***.

Stack and Stack Frames

The ***stack*** grows and shrinks linearly as functions are called and returned.

For Linux/x86 and many UNIX implementations, the stack resides at the high end of the virtual address space and ***grows downwards*** towards the ***heap region***:

- Each time a function is called, a new stack frame is allocated on the stack
- The top of the stack is pointed to by a special-purpose register, known as the ***stack pointer***.

Stack and Stack Frames

Each ***stack frame*** contains the following information:

- Function arguments and local variables:
  - In C/C++, these variables are referred to as ***automatic*** variables since they are automatically created when a function is called and automatically destroyed when the function returns.
- Call linkage information:
  - Each function uses certain ***CPU registers***, such as the program counter, which points to the next machine instruction to be executed.
  - Each time a function calls another, a copy of these registers is saved on the ***callee***'s stack frame so that when the function returns, the appropriate register values can be restored for the ***caller***.

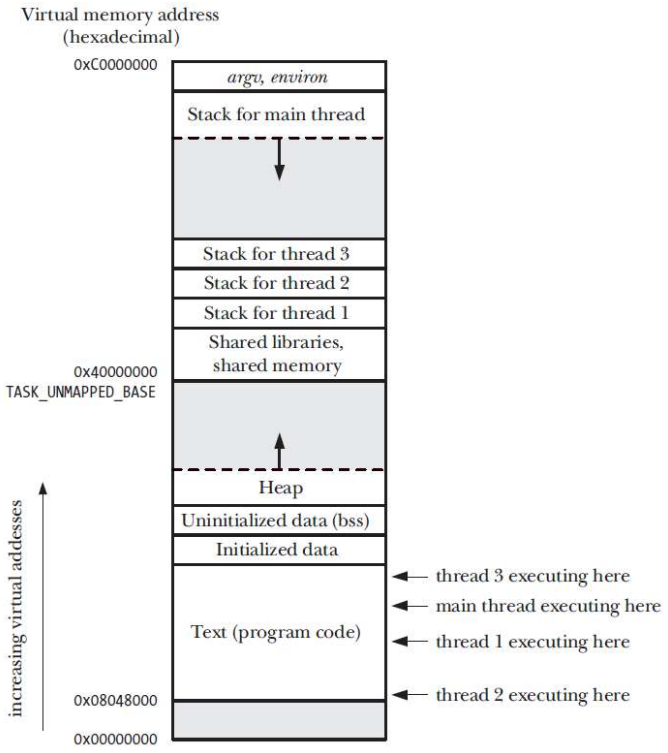
Threads

Like processes, threads are a mechanism that allows an application to execute concurrently.

A single process can contain multiple threads, all of which concurrently execute the same program (***text segment***) and share the ***same virtual address space***.

- The threads in a process can execute concurrently.
- On a multiprocessor system, multiple threads can execute in parallel.
- If one thread is blocked on an I/O operation, other threads can still proceed and execute.

Threads



Typical memory layout of a multithreaded process on Linux/x86-32

Threads

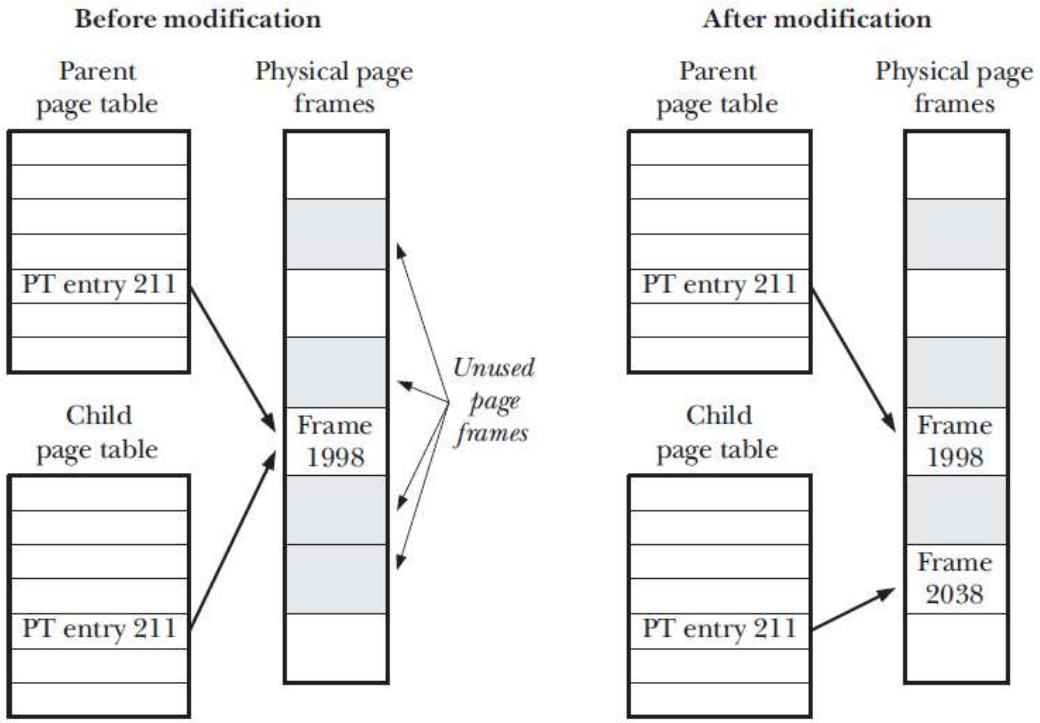
Process:

- It is difficult to share information among a multiprocess application even though the processes were created through the ***fork*** system call.
  - The parent and the child do not share the virtual address space although they may share the ***text segment*** so some form of IPC must be used to permit the processes in the application to communicate.
- The creation of a process with the ***fork*** system call is relatively expensive even with the ***copy-on-write*** (***COW***) technique since there must be a separate set of resources such as a page table, a file descriptor table etc.

Thread:

- Sharing information between threads is easy and fast since it is just a matter of reading and writing shared variables (***data segment+heap segment***).
- The creation of a thread is faster than that of a process through the ***clone*** system call – typically an order of magnitude faster – since they all share the same virtual address space; hence ***COW*** is not required.

Copy-on-Write



Page tables before and after modification of a shared copy-on-write page

Synchronization Primitives

Tasks frequently need to communicate with each other.

The following ***three questions*** can arise:

1) How do we share data among each other?
2) How do we make sure that two or more tasks do not get in each other way?
3) How do we make sure the proper sequencing when dependencies between them are present?

We will discuss 1) in details in the lectures on ***shared-memory programming***.

Let us now focus on 2) and 3), which involves ***mutual exclusion*** and ***synchronization*** issues, respectively.

Race Conditions

Threads that are working together may share some common storage each one can read and write.

- The shared storage may be in main memory or it may be a shared file.

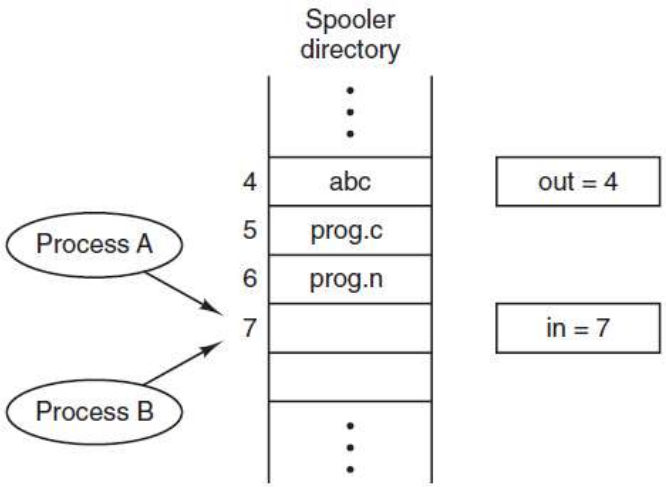
Let us consider a simple example: a ***printer spooler***.

- When a process (a user) wants to print a file, it enters the file name in a special ***spooler directory***.
- Another process, the ***printer daemon***, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.

Race Conditions: Printer Spooler

Imagine that

- our ***spooler directory*** has a very large number of slots, numbered 0, 1, 2, … each one capable of holding a file name.
- there are ***two shared variables***, out, which points to the next file to be printed, and in, which points to the next free slot.



Race Conditions: Printer Spooler

At a certain instant, slots 0 to 3 are empty and slots 4 to 6 are full.

Suppose that, more or less simultaneously, ***Process A*** and ***Process B*** are queuing file for printing and the following could happen.

- ***Process A*** reads in and stores the value of 7 in a local variable called $next\_free\_slot$.
- Just then a clock interrupt occurs, and the OS scheduler decides that it should context-switch to ***Process B***.
- ***Process B*** also reads in and also gets 7 and stores this value into its local variable $next\_free\_slot$.
- At this moment, both think that the next available slot is 7.
- ***Process B*** continues to run, stores the name of the file in slot 7, and updates in to 8.
- Eventually, ***Process A*** runs again, starting from where it left off.
- ***Process A*** looks at $next\_free\_slot$, finds a 7 there, writes its file name in slot 7, effectively erasing the name that ***Process B*** just put there, and computes $next\_free\_slot + 1$, which is 8 so it sets in to 8.

Race Conditions: Printer Spooler

What has gone wrong?

- The ***printer daemon*** will not notice anything wrong, but ***Process B*** will never get its file printed.

Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called ***race conditions***.

Critical Section

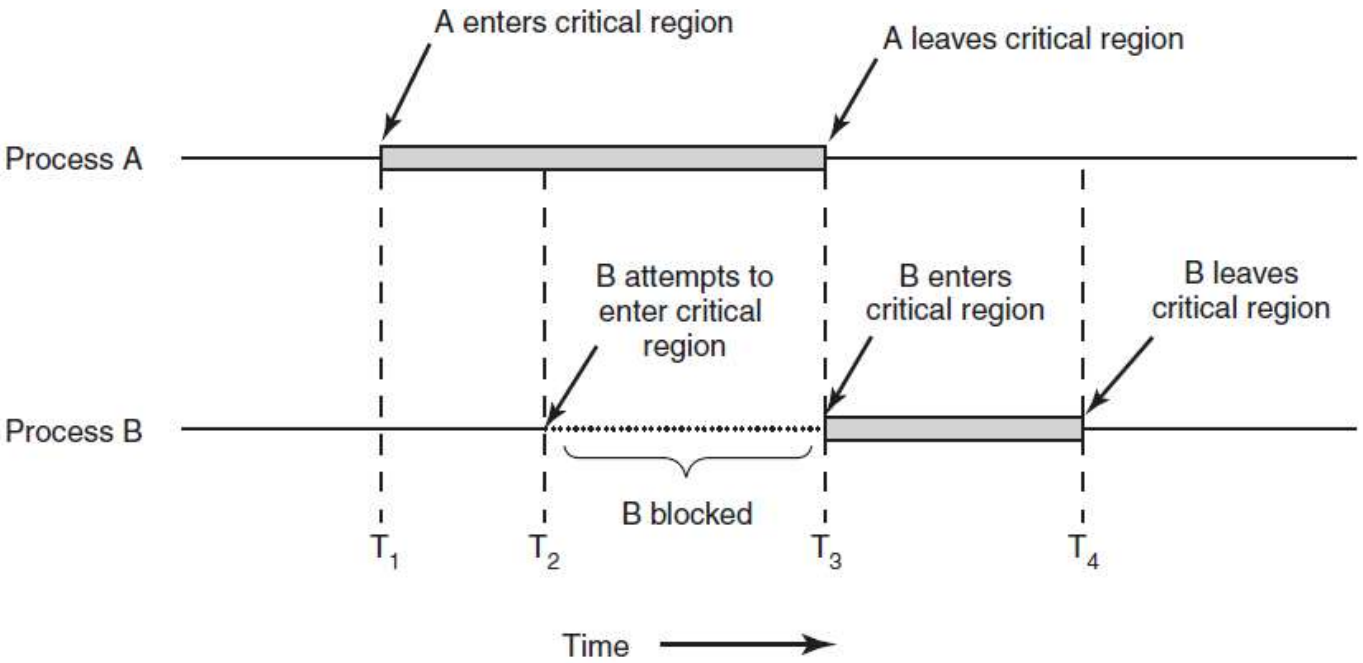
How do we avoid ***race conditions***?

- We must find a way to prohibit more than one process from reading and writing shared data at the same time.
- In other words, what we need is ***mutual exclusion***.

The problem of avoiding race conditions can also be formulated in a general way as follows:

- Part of the time, a process is busy doing local computations and other things that do not lead to race conditions.
- However, sometimes, a process has to access shared memory or files, which can lead to races.
- That part of the program where the shared memory is accessed is called a ***critical section***.

Critical Section



Mutual Exclusion: Disabling Interrupts

On a ***uniprocessor*** system, the simplest solution is to have each process

- disable all interrupts just after entering its critical section
- re-enable them just before leaving it

With interrupts disabled, no clock interrupts can occur to trigger a ***context switch*** to another process.

- Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other processes will intervene.
- It is unwise to give user processes the power to turn off interrupts.
  - What if one of them did it, and never turned them on again?
  - That could be the end of the system.
- Moreover, disabling interrupts cannot avoid race conditions on ***multiprocessor*** system.
  - Disabling interrupts affects only the CPU that executed the *CLI* instruction.
  - Disabling interrupts is often a useful technique within the OS kernel itself but it is not appropriate as a general mutual exclusion mechanism for user processes.

Mutual Exclusion: Spinlocks

The TSL Instruction:

Modern computers are multi-core processors that come with an instruction like:

```
TSL RX,LOCK
```

The ***Test-and-Set*** (***TSL***) instruction works as follows:

- It reads the content of the memory word $LOCK$ into register RX and stores a value of 1 into memory address $LOCK$.
- The operations of reading the word and storing into it are guaranteed to be ***indivisible***, i.e., no other processors can access the memory word until the ***TSL*** instruction is finished.
- The processor executing the ***TSL*** instruction locks the memory bus to prohibit other processors from accessing memory until it is done.

Mutual Exclusion: Spinlock

The TSL Instruction:

```
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 not 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 Exclusion: Spinlock

An alternative to ***TSL*** is ***XCHG***, which exchanges the contents of two locations atomically, for example, a register and a memory word.

- All Intel x86 CPUs use the ***XCHG*** instruction for low-level synchronization.

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
        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 Exclusion: Spinlock

A spinlock that uses ***TSL*** tends to generate ***high coherence traffic****:*

- When each thread spins on a lock using TSL, it writes a value of 1 to the memory location of the lock.
- The write ***invalidates*** the ***cache line*** of the lock cached on other CPU cores.
- Therefore, if multiple threads are simultaneously spinning on the same lock, the cache coherence protocol will generate a large number of cache invalidation messages.

Mutual Exclusion: Spinlock

This issue can be mitigated by using the ***Test-and-Test-and-Set*** technique, which relies on the fact that ***reading*** does not generate coherence traffic if the lock is cached locally.

```
lock:   test   register, location
        bnz    register, lock
        t&s    register, location
        bnz    register, lock
        CS
        st     location, #0
```

Mutual Exclusion: Spinlock

When a thread enters $enter\_region$ and fails to acquire the lock, it simply waits in a tight loop or "***spin***" while repeatedly checking if the lock is free.

- Since the CPU still actively executes instructions but does not perform a useful computation, we refer to this as "***busy waiting***".
- Use of spinlocks makes no sense in uniprocessor systems.
  - A thread waiting on a lock should immediately ***yield*** the CPU so that another thread can run and will release the lock if it is the one currently holding the lock.
- Generally, we choose to use a spinlock for the following reasons:
  - The ***critical section*** is ***short***.
  - There is ***low contention*** for the lock locally.

Mutual Exclusion: Spinlock

Spin vs Sleep and Wake up:

- Spinning wastes CPU cycles for no useful task.

- Sleep and Wake up, which means yielding the CPU and context-switching to a new process or thread, incurs a relatively high overhead for the following reasons:
  - ***Saving and restoring contexts*** are expensive
  - ***Cache working sets*** will change across context switches
  - ***TLB*** needs to be flushed if a different process is picked by the OS scheduler.
    - Modern CPUs support ***Process-Context Identifiers***.

Generally, if the lock is held for a period of time smaller than the overhead of context-switching, it is better to use ***spinlocks***.

Mutual Exclusion: Spinlock

Spinning Approaches:

- Keep spinning until the lock is acquired
- Spin only for some amount of time and yield if the lock is not acquired.

```
void mutex lock (volatile lock t *l) {
  while (1) {
      for (int i=0; i<MUTEX N; i++)
            if (!test and set(l))
                  return;
      yield();
  }
}
```

Mutual Exclusion: Spinlock

Spin and Yield:

Spin only for some amount of time and yield if the lock is not acquired

- The spin time should be long enough for the CPU owning the lock to exit the critical section
- This strategy is useful if the ***expected spin time*** is less than the number of iterations N.

Mutual Exclusion: Spinlock

Disable Interrupts and Spin:

Disable interrupts before spinning and re-enable interrupts after releasing the lock.

- Efficient for short critical sections
- Not appropriate for user-mode processes or threads

```
void mp spinlock (volatile lock t *l) {
   cli(); // prevent preemption
   while (test and set(l)) ; // lock
}
void mp unlock (volatile lock t *l) {
   *l = 0;
   sti();
}
```

Sleep and Wake up

In stead of spinning in a tight loop waiting on the lock, a process or a thread can, instead, ***yield*** by putting itself to ***sleep*** and thereby trigger a ***context switch***.

There are a few ***blocking synchronization primitives*** we will discuss.

- ***Semaphore***
- ***Mutex***
- ***Condition Variable***
- ***Monitor***

Mutex

A ***mutex*** is a shared variable that can be either one of the two states:

- ***Locked***
- ***Unlocked***

Therefore, a mutex is used to provide ***mutual exclusion*** to shared resource.

Each mutex has two basic operations associated with it:

- $mutex\_lock$
  - When a thread wants to enter a critical section, it calls $mutex\_lock$.
  - If the mutex is currently unlocked, the thread succeeds and is free to enter the critical section.
  - On the other hand, if the mutex is already locked, the calling thread is blocked until the thread that is in the critical section is finished and calls $mutex\_unlock$.
- $mutex\_unlock$
  - When a thread exits a critical section, it releases the mutex.
  - If multiple threads are blocked waiting on the mutex, one of them is chosen at random and allowed to acquire the mutex.

Mutex: Implementation

```
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
ok:     RET                     | return to caller; critical region entered

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

Mutex: Mutex vs Spinlock

The implementation of $mutex\_lock$ is similar to that of $enter\_region$ but with ***one crucial difference***.

When $enter\_region$ fails, it keeps spinning (busy waiting).

When $mutex\_lock$ fails, it calls $thread\_yield$ to give up the CPU to another thread.

Semaphore

A ***semaphore*** is a synchronization primitive that can be used as an ***atomic counter***.

- Provides two basic operations, namely, P and V.
  - P for decrementing the integer value of the semaphore
  - V for incrementing the integer value of the semaphore
- Associated with each semaphore is an integer value operated on by the two basic operations, P and V.
  - Initially, a semaphore needs to be initialized to an integer value n.
  - At any moment, the value cannot be larger than n.
- Associated with each semaphore is a queue of ***waiting processes***.
  - The queue is maintained by the OS kernel
  - P and V are implemented as ***system calls***

In this lecture, let us refer to P as $wait$ and V as $signal$, respectively.

Semaphore: Basic Operations

Given a ***semaphore*** s,

- $wait(\mathrm{s})$ decrements the value of s by one.
  - If the new value is ***negative***, the process that performed $wait(\mathrm{s})$ is blocked and is added to the semaphore queue.
  - Otherwise, the process continues execution.
- $signal(\mathrm{s})$ increments the value of s by one.
  - If the new value after the increment is still ***non-positive***, this means that there are still processes waiting for a resource and the OS removes one waiting process from the semaphore queue and makes the process ready by putting into the ready queue.

Semaphore: Implementation

```
wait (S) {
  Disable interrupts;
  while (S->value == 0) {
    enqueue(S->q, current_thread);
    thread_sleep(current_thread);
  }
  S->value = S->value – 1;
  Enable interrupts;
}
```

```
signal (S) {
  Disable interrupts;
  thread = dequeue(S->q);
  thread_start(thread);
  S->value = S->value + 1;
  Enable interrupts;
}
```

Semaphore: Types

Semaphores come into two types:

- ***Mutex or Binary Semaphore***
  - represents a single access to a shared resource
  - guarantees a mutual exclusive access to the resource
- ***Counting Semaphore***
  - represents a shared resource with ***multiple units*** available or a resource that allows certain kinds of unsynchronized concurrent access, e.g. , reading
  - Multiple threads can execute past a $wait(\mathrm{s})$ operation at any moment.

Semaphore: Mutual Exclusion Problem

```
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```

Semaphore: Readers-Writers Problem

Problem: An object is shared among threads of two kinds:

- ***Readers*** only read the object.
- ***Writers*** write to the object.

Concurrent reads are allowed but writes must be performed on the object in a mutual exclusive manner.

We can use ***two binary semaphores*** to control access to the object:

- Semaphore $mutex$
  - Associated with $mutex$ is a variable $readcount$ for counting the number of concurrent reads.
  - $mutex$ controls mutual exclusive access to $readcount$
- Semaphore w_or_r
  - Provides exclusive writing or reading
  - At any moment, there can be only one kind of accesses to the object

Semaphore: Readers-Writers Problem

```
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}
```

```
reader {
    wait(mutex);       // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex);   // unlock readcount
    Read;
    wait(mutex);       // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex);   // unlock readcount}
}
```

Semaphore: Readers-Writers Problem

Observations:

- If there is a writer currently writing, the first reader blocks waiting on $signal(w\_or\_r)$, while the remaining readers block on $signal(mutex)$.
- Once a writer is done writing, all readers can fall through.
- The last reader signals a waiting writer.
  - If there is no writer, readers can continue.
- Notice that if there are writers and readers blocked on $signal(w\_or\_r)$ and a writer exits the critical section, it is more likely that the next waiting readers will win.
  - This is because writers must acquire $signal(w\_or\_r)$ individually whereas if one reader acquires $signal(w\_or\_r)$, all readers will fall through.
  - Starvation can potentially occur as the algorithm prefers readers to writers.
  - Other algorithms exist that addresses the fairness issue.

Semaphore: Bounded Buffer Problem

Problem: There is a buffer of bounded size N shared by a producer and a consumer thread.

- The producer ***adds*** resources into the buffer.
- The consumer ***removes*** resources from the buffer.

In this problem, we can use ***three semaphores***:

- Semaphore $mutex$
  - Provides mutual exclusive access (add/remove operation) to the buffer
  - initialized to 1
- Semaphore $full$
  - Counts the number of full slots
  - initialized to 0
- Semaphore $empty$
  - Counts the number of empty slots
  - initialized to N

Semaphore: Bounded Buffer Problem

```
Semaphore mutex = 1;   // mutual exclusion to shared set of buffers
Semaphore empty = N;   // count of empty buffers (all empty to start)
Semaphore full = 0;    // count of full buffers (none full to start)
```

```
producer {
  while (1) {
    Produce new resource;
    wait(empty); // wait for empty buffer
    wait(mutex); // lock buffer list
    Add resource to an empty buffer;
    signal(mutex); // unlock buffer list
    signal(full);      // note a full buffer
  }
}
```

```
consumer {
  while (1) {
    wait(full);        // wait for a full buffer
    wait(mutex);    // lock buffer list
    Remove resource from a full buffer;
    signal(mutex); // unlock buffer list
    signal(empty); // note an empty buffer
    Consume resource;
  }
}
```

Semaphore: Bounded Buffer Problem

Observations:

- We make use of semaphores in two different ways:
  - The $mutex$ semaphore is used for ***mutual exclusion***.
    - It guarantees that at most one process at a time can access or modify the state of the buffer.
  - The counting semaphores $full$ and $empty$ are used for ***synchronization*** by ensuring that certain event sequences do or do not occur.
    - It makes sure that the producer must wait when the buffer is full.
    - It makes sure that the consumer must wait when the buffer is empty.

Monitor

A ***monitor*** is a OOP-style construct that provides structured synchronized access to shared data:

- Encapsulates data to be shared between threads
- Provides methods for operating on those shared data in such a way that ***at most one thread*** can call a method and be inside the monitor at a time – hence mutual exclusion
  - Methods define ***critical sections***.

In effect, a monitor protects its shared data from unstructured accesses through their "***monitor methods***".

Monitor

A monitor provides ***mutual exclusion***:

- Only one thread can execute any monitor method at a time.
  - We say that the thread is "***in the monitor***".
- If a second thread attempts to call a monitor method, when a first thread is already in the monitor, the second thread is blocked.
  - The monitor will add the blocked thread to the ***ready queue***.
- If the thread in the monitor is blocked, a different thread can enter the monitor.

A monitor also provides ***synchronization*** through the use of ***condition variables***, which we will talk about in the next few slides.

Monitor: Condition Variable

A ***condition variable*** provides a mechanism to wait for ***certain events***:

- e.g. buffer is not empty, buffer is not full etc.

Each condition variable has a ***waiting queue*** for queuing threads waiting on the corresponding events to occur.

Condition variables support ***three basic operations***:

- $wait$ releases the monitor lock, wait for a condition variable to be signaled in the waiting queue of the condition variable.
- $signal$ wakes up one thread.
- $broadcast$ wakes up all waiting threads.

Monitor: Bounded Buffer Problem

```
Monitor bounded_buffer {
  Resource buffer[N];
  // Variables for indexing buffer
  Condition not_full, not_empty;

  void put_resource (Resource R) {
    while (buffer array is full)
        wait(not_full);
    Add R to buffer array;
    signal(not_empty);
  }
```

```
  Resource get_resource() {
    while (buffer array is empty)
        wait(not_empty);
    Get resource R from buffer array;
    signal(not_full);
    return R;
  }
} // end monitor
```

Monitor: Mechanism

Access to a ***monitor*** is controlled by a ***lock*** to provide mutual exclusive access to the protected shared data inside the monitor:

- $wait$ blocks the calling thread and gives up the ***lock.***
  - At this point, another thread can acquire the lock and enter the monitor.
- $signal$ causes one arbitrary waiting thread to wake up.
  - If there is no thread waiting on the condition variable, the signal is lost.
- $broadcast$ causes all waiting threads to wake up.

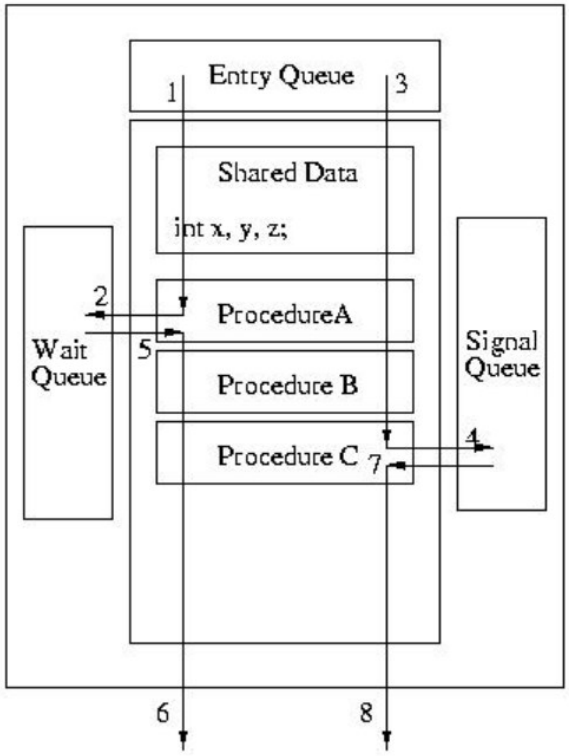
These ***three basic operations*** can be called only when a thread is already inside the monitor.

Monitor: Signaling Semantics

There exist two basic types of ***signaling semantics*** for monitors.

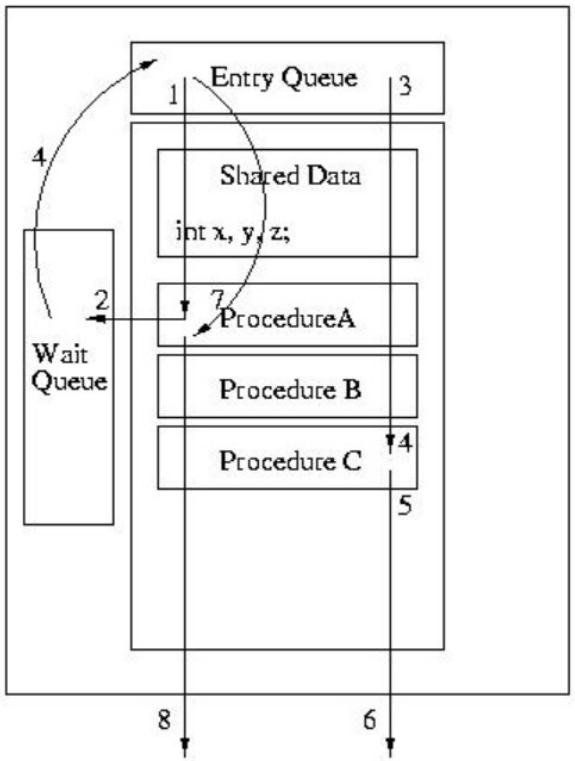
- ***Hoare***
  - $signal$ immediately passes the monitor lock from the signaler to one waiter thread in the waiting queue of the condition variable
  - The monitors adds the signaler to the ***signal queue***.
  - The waiter runs immediately, which means that the condition the waiter was waiting for is guaranteed to hold when the waiter executes: an ***if-statement*** is used.
  - After the waiter is done, a thread from the signal queue is restarted.
  - If the signal queue is empty, schedule one from the ready queue.
- ***Mesa***
  - $signal$ places removes one waiter from the waiting queue of the condition variable and add it to the ready queue of the monitor, but the signaler continues to execute inside the monitor.
  - The condition the waiter was waiting for does not still necessarily hold when the waiter runs so the waiter must recheck the condition: an ***while-statement*** is used.

Monitor: Hoare



1. ThreadA enters the monitor
2. ThreadA waits for a resource
3. ThreadB enters the monitor
4. ThreadB signals on the resource and enters the signal queue
5. ThreadA re-enters the monitor
6. ThreadA leaves the monitor
7. ThreadB re-enters the monitor
8. ThreadB leaves the monitor
9. Another process can be admitted form the entry queue

Monitor: Mesa



1. ThreadA enters the monitor
2. ThreadA waits for a resource
3. ThreadB enters the monitor
4. ThreadB signals on the resource; ThreadA moves to the entry queue
5. ThreadB continues in the monitor
6. ThreadB leaves the monitor
7. ThreadA moves from the entry queue back into the monitor
8. ThreadA leaves the monitor
9. Another process can be admitted form the entry queue

Monitor: Signaling Semantics

Trade-offs:

Hoare Semantics:

+ More deterministic, hence easier to reason about
- Less efficient due to more context switches
- Broadcast is hard

Mesa Semantics:

+ less deterministic
+ Broadcast is easy
- More efficient due to fewer context switches

References

[1] Andrew S. Tanenbaum and Herbert Bos. 2014. Modern Operating Systems (4th. ed.). Prentice Hall Press, USA.

[2] Michael Kerrisk. 2010. The Linux Programming Interface: A Linux and UNIX System Programming Handbook (1st. ed.). No Starch Press, USA.