Note: this is combining a ton of the previous lecture notes so that all the notes from the `Concurrency: Mutual Exclusion and Synchronization` slide deck are in one file

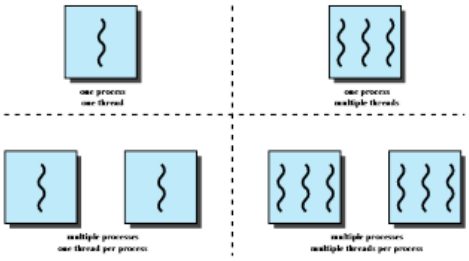
We'll start in the Monitors section.

toc:

Recap - Processes and Threads Management

The central themes of operating system design are all concerned with the management of processes and threads:

1. **Multiprogramming:** The management of multiple processes within a uniprocessor system.
2. **Multiprocessing:** The management of multiple processes within a multiprocessor.
3. **Distributed processing:** The management of multiple processes executing on multiple, distributed computer systems (e.g., clusters).

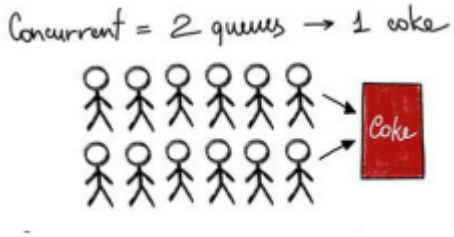


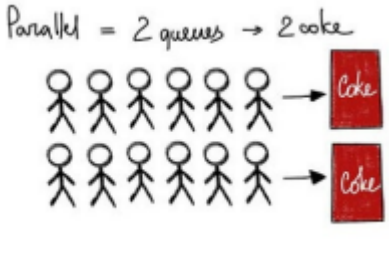
Threads and Processes

Background

processes can execute concurrently or in parallel

may be interrupted at any time, partially completing execution





we have a shared memory where we store variables and everything we want to work with.

If we don't manage that shared memory properly we end up with data inconsistency

- Maintaining data consistency requires mechanisms to ensure the orderly execution of **cooperating** processes that share a logical address space.
  - A process that can affect or be affected by other processes executing in the system.

make sure the processes execute in an orderly fashion and don't fuck with each other when they share a resources

Race Condition

- several processes access and manip the same data at the same time
- outcome depends on who got there first

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Critical-Section Problem

critical segement of code:

- process may be accessing and updating data that is shared with at least one other process
- only one process in critical section at a time

when a process wants to enter the critical section

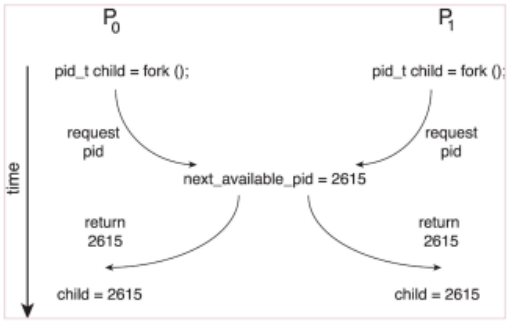
- entry section - process asks permission to enter
- it does the critical section
- exit section - let's eeryone know it's done
- remainder section is the rest of the noncritical stuff

we need to ensure the 3 following conditions

1. mutual exclusion
   - only one process in critical section at a time
2. progress
   - if no one else is in critical and someone else wants to go critical then we have to select the next process
   - we can't postpone this selection indefinitely
3. bounded waiting
   - there must be a limit on the number of times that other processes can enter the critical sections after a process has made a request to enter
   - assume same speed of all processes

Kernel-Mode Race Condition Example

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (`pid`)



- Unless there is mutual exclusion, the same pid could be assigned to two different processes!

Single Core

very shrimple

we could prevent interrupts from occuring whilue a shared variable was being modified

useless when we try to move to multiprocessor since it would just start disabling everything.

OS

2 approaches depending on preemption

- preemptive
  - allows preemption of process when running in kernel mode
  - processor can tell the process to pack all their bags and get ready to move
- non-preemptive
  - runs until kernel mode, blocks, or voluntarily yields CPU
  - essentially free of race conditions as only one process is active in the kernel at a time

preemptive are difficult to design.

it allows two kernel-mode processes to run simultaneously on diff cores

it also makes things more responsive

---

knowledge check

- A race condition _____.
  a) results when several threads try to access the same data concurrently
  b) results when several threads try to access and modify the same data concurrently
  c) will result only if the outcome of execution does not depend on the order in which instructions are executed
  d) None of the above

- Instructions from different processes can be interleaved when interrupts are allowed.
  - True
  - False

- A(n) _____ refers to where a process is accessing/updating shared data.
  a) critical section
  b) entry section
  c) mutex
  d) test-and-set

- A non-preemptive kernel is safe from race conditions on kernel data structures.
  - True
  - False

- A solution to the critical section problem does not have to satisfy which of the following requirements?
  a) mutual exclusion
  b) progress
  c) atomicity
  d) bounded waiting

Atomic Operation

- A function or action implemented as a sequence of one or more instructions that appears to be **indivisible**:
  - No other process can see an intermediate state or interrupt the operation.
  - The sequence of instruction is guaranteed to execute as a group, or not execute at all.

atomic operations

Mutual Exclusion

Software Approaches

Dekker's Algo

Dijkstra went to go publish it for him so that's why there's a jpeg of him on the slide

first known sol'n to the mutual exclusion problem in concurrent programming

allows 2 threads to shar ea single-use resource without conflict, using on ly shared memory for communication

only one access to a memory location can be made at a time

we have a global memory location labelled `turn` that is shared between the two processes

Processes check to see if it's their turn and execute accordingly. Once their done they give the turn to the other process.

Mutual exclusion is guaranteed.

What is the problem?

---

prof left off here for 05/02/24 lect

---

midterm is all question formats and on paper

Porcesses must strictly alternate in their use of their critical section

- the pace of execution is dictated by teh slower of the two processes

One process being slow makes the whole things slower since everyone has to wait on that slower process.

if one process fails then the other process is permanently blocked

- this happens because the failed process can't give the turn over to the other process by exiting and setting the flag
- this happens whether or not the process is in the critical section or not

2nd attempt - a revision:

- each process has it's **own key *(flag)***
- if one fails, the other can still access its critical section
- processes can look at others flags
- one process can't change the flag of another process
- processes share a boolean vector that holds the flags
  - `Boolean vector flag[2]`

```
   /* PROCESS 0 *              /* PROCESS 1 *
.                           .
.                           .
while (flag[1])             while (flag[0])
   /* do nothing */;           /* do nothing */;
flag[0] = true;             flag[1] = true;
/*critical section*/;       /* critical section*/;
flag[0] = false;            flag[1] = false;
.                           .
```

drawbacks:

- permanent blocking still possible
  - process fails inside critical section or after setting flag to true w/o entering
- no guaranteed mutual exclusion
  - P0 exeutes the while statement and finds flag[1] set to false
  - P1 executes the while statement and finds flag[0] set to false
  - P0 sets flag[0] to true and enters its critical section
  - P1 sets flag[1] to true and enters its critical section
  - now both processes arenow executing their critical sections so the program is incorrect

3rd attempt - another revision

- the problem from before was that a process can change its state after the other process has checked it but before the other process can enter its critical section
- swap around the two stateemnts to guarantee mutual exclusion

```
   /* PROCESS 0 *              /* PROCESS 1 *
.                           .
.                           .
flag[0] = true;             flag[1] = true;
while (flag[1])             while (flag[0])
   /* do nothing */;           /* do nothing */;
/* critical section*/;       /* critical section*/;
flag[0] = false;            flag[1] = false;
.                           .
```

we set the flag's value before we execute the while loop

drawbacks:

- this results in a deadlock

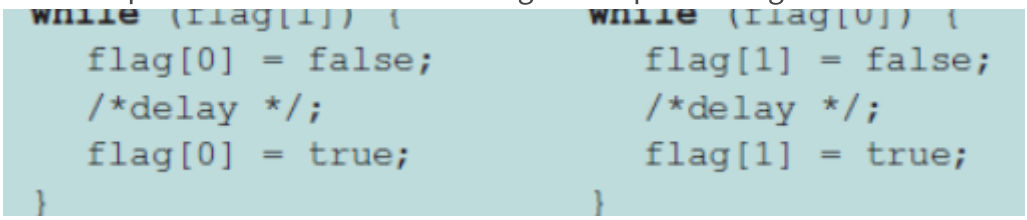
- both processes set their flags to true
- later on they will check each other's flags and decide they need to wait for the other
  - the flag value makes them think the other process is in their critical section

4th attempt - yet another revision:

- Each process sets its flag to indicate its desire to enter its critical section but is prepared to reset the flag to defer to the other process
- mtuual exclusion is guaranteed under this model

```
   /* PROCESS 0 *                /* PROCESS 1 *
.                             .
.                             .
flag[0] = true;               flag[1] = true;
while (flag[1]) {             while (flag[0]) {
   flag[0] = false;              flag[1] = false;
   /*delay */;                   /*delay */;
   flag[0] = true;               flag[1] = true;
}                             }
/*critical section*/;         /* critical section*/;
flag[0] = false;              flag[1] = false;
.
```

drawbacks:

- 
  - each process sets their flag to true
- 
  - each process checks the other's flag before proceeding with the while loop
- 
  - each flag flips from false and back to true
- the loop ends up repeating in what's known as a *livelock*
- this won't sustain for a long time
  - the processes are different and we can presume that the while loops won't always take the same amount of time and that one will finish their while loop before the other can so they'll enter the critical section first

5th iteration - the correct solution:

- we introduce ***the right to insist***

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          while (flag [1]) {
               if (turn == 1) {
                    flag [0] = false;
                    while (turn == 1) /* do nothing
*/;
                    flag [0] = true;
               }
          }
          /* critical section  */;
          turn = 1;
          flag [0] = false;
          /* remainder   */;
     }
}
void P1( )
{
     while (true) {
          flag [1] = true;
          while (flag [0]) {
               if (turn == 0) {
                    flag [1] = false;
                    while (turn == 0) /* do nothing
*/;
                    flag [1] = true;
               }
          }
          /* critical section   */;
          turn = 0;
          flag [1] = false;
          /* remainder   */;
     }
}
void main ()
{
     flag [0] = false;
     flag [1] = false;
     turn = 1;
     parbegin (P0, P1);
}
```

- when P0 wants to enter the critical section
  - it sets its flag to true
  - checks P1's flag
    - if P1 flag is false then P0 gets to go critical
    - else, P0 consults turn
      - turn==0
        - it's P0's turn to insist
        - P0 will just keep on checking P1's flag in order to try and enter its critical section
- P1 will at some point note that it is its turn to defer and set its flag to false, allowing P0 to proceed
- after P0 has used its critical section
  - P0 flag set to false
  - turn set to 1 to transfer the right to sinsit to P1

Peterson's Solution

Dekker's algo solves things but it's hard to follow

Peterson has a simpler sol'n that allows two processes to share a single-use resource w/o conflict

only shared memory is used for communication

it is kind of rewriting Dekker's algorithm

```
bool flag[2] = {false, false};
int turn;
```

```
P0:      flag[0] = true;
P0_gate: turn = 1;
         while (flag[1] && turn == 1)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[0] = false;
```

```
P1:      flag[1] = true;
P1_gate: turn = 0;
         while (flag[0] && turn == 0)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[1] = false;
```

`int turn` - indicates whose turn it is to enter the critical section

`bool flag[2]` - indicate if a process is ready to enter the critical section

`flag[i]==true` implies that process P_i is ready

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;
      /* critical section */
        flag[i] = false;
      /* remainder section */
}
```

3 critical-section requirements are met(provably):

1. mutual exclusion is preserved
   - P_i enters critical section only if: either `flag[j]==false` or `turn == i`
     - the other process is not ready or
     - it is P_i's turn
2. Progress requirements is satisfied
3. Bounded-waiting requirement is met

Useful for demonstrating an algo but not guaranteed to work on modern architectures.

to improve performance, processors and/or compilers may reorder operations that have no dependencies

- fine for single-threaded as result is always the same
- may produce inconsistent or unexpected results for multithreading

```
two threads share _flag_ and _x_ variables

boolean flag = false;
int x = 0;

Thread 1 performs

while (!flag)
  print x;

Thread 2 performs
```

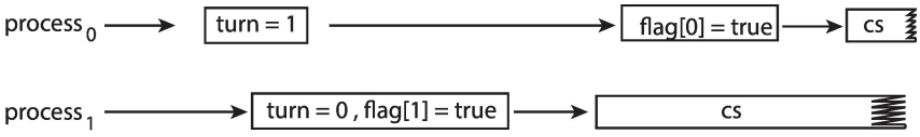
```
x = 100;
flag = true;

what is the expected output

100 is the expected output however the operations for thread 2 may be reordered so the output might be 0
```

Original

```
while (true){
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;
        /* critical section */

        flag[i] = false;
        /* remainder section */
 }
```

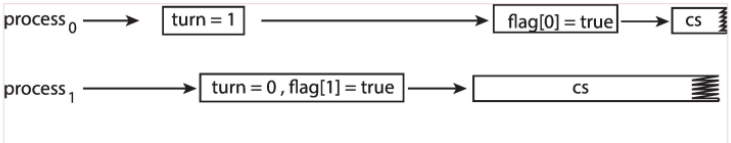
Reordered

```
while (true){
        turn = j;
        flag[i] = true;
        while (flag[j] && turn = = j)
                ;
        /* critical section */

        flag[i] = false;
        /* remainder section */
 }
```



- Process 0: turn = 1;
- Process 1: turn = 0;
- Process 1: flag[1] = true;
- Process 1: while(flag[0] && turn==0) // terminates, since flag[0]==false
- Process 1: enter critical section 1
- Process 0: flag[0] = true;
- Process 0: while(flag[1] && turn==1) // terminates, since turn==0
- Process 0: enter critical section 0

```
while (true){
        turn = j;
        flag[i] = true;
        while (flag[j] && turn = = j)
                ;
        /* critical section */
        flag[i] = false;
        /* remainder section */
 }
```



We have to use a _**Memory Barrier** to make sure that Peterson's solution will work correctly on modern computer architecture.

So this software based support will need the aid of some hardware.

Knowledge check:

- In Peterson's solution, the ___ variable indicates if a process is ready to enter its critical section
  - `flag[i]`

Hardware Support Approaches

“There is no bullet solution… all solutions build on each other”

Many systems provide hardware support for implementing the cirtical section code

uniprocessors - could disable interrupts

- currently running code would execute w/o preemption
- genearlly, too inefficient on multiprocessor systems
  - OS using this are not broadly scalable

looking at 3 forms of hardware support

1. memory barriers (aka memory fences)
2. hardware instructions
3. atomic variables

Memory Barrier

How a computer architecture determines what memory guarantees it will provide to an application program is known as its memory model.

models may be either

- strongly ordered
  - memory modification of one processor is immediately visible to all other processors
- weakly ordered
  - not always immediately visible to all other processors

developers cannot make an assumption about whether or not the memory model is strongly or weakly ordered.

in lieu of this guarantee, we use memory barriers

memory barriers

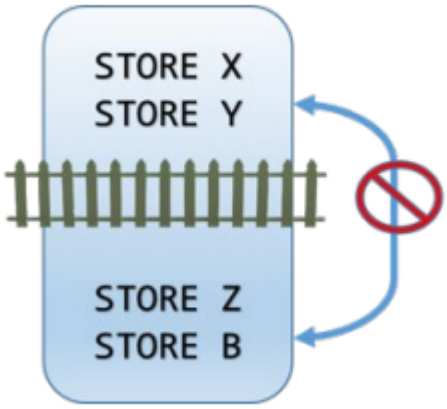
- computer instructions that force any changes in memory to be propogated to all other processors in the system

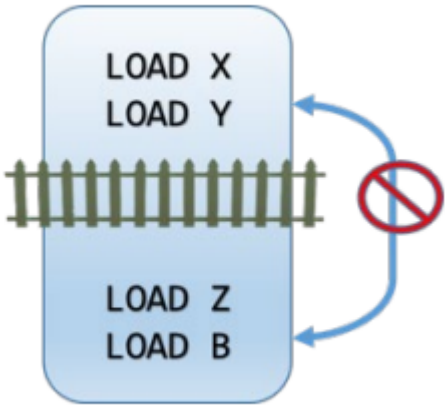
`Sfence` is a memory barrier instruction for example with regards to storing

`lfence` is another type of instruction with regards to loading

`mfence` prevents reording of reading and writing before or after the fence

do not reorder anything before the barrier with anything that comes after the barrier





5/07/10/memory-barriers-in-dot-net.aspx/

we're not able to mov the loads to either side of the fence

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:

- **Thread 1** now performs

```
while (!flag)
     memory_barrier();
print x
```

- **Thread 2** now performs

```
x = 100;
memory_barrier();
flag = true
```

- For Thread 1 we are guaranteed that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

Hardware Instructions

```
boolean test_and_set (boolean *target){
  boolean rv = *target;
  *target = true;
  return rv;
}
```

`test_and_set()`

- executed atomically
- read a value in memory
- set it to true
- return the value that was in memory

This let's us make a lock out of the flag we want us to make sure we don't enter the critical section while anothre program is in its critical section.

```
do {
  while (test_and_set(&lock));
  /*critical section*/
  lock = false;
  /*remainder section*/
} while (true);
```

This is suported by nearly every architecture.

`compare_and_swap()`

```
int compare_and_swap(int *value, int expected, int new_value) {
  int temp = *value;
  if (*value == expected)
    *value = new_value;
  return temp;
}
```

- executed atomically
- returns the original value of `value`
- set the `value` to the value of `new_value`
  - but only if `*value == expected`

We can also use this to make a shared lock for mutual exclusion. Have it initialized to 0.

```
while (true) {
  //lock is the lock, 0 is the expected value, 1 is the prospective new value
  while (compared_and_swap(&lock, 0 ,1) != 0) {/*do nothing, someone else is in
critical section*/}
  /*critical section, lock has been set to by us*/
  lock = 0;     //we left the critical section so we're able to set the lock back to
0
  /*remainder section*/
}
```

Atomic Variables

Typically instructions such as compare-and-swap are used as bhilding blocks for other synchronization tools.

One tool is an atomic variable that provdes *atomic* (uninterruptible) updates on basic data types such as integers and booleans.

Ex:

- `sequence` - our atomic variable
- `increment()` - an operation on `sequence`
- command
  - `increment(&sequence)`
  - ensures the sequence is inremented without interruption

we make atomic updates so that we're guaranteed to update a variable.

`increment()` can be implemented like so:

```
void increment (atomic_int *v) {
  int temp;
  do {
    temp = *v;
  }
  while (temp != compare_and_swap(v, temp, temp+1));
  // the above keeps looping until we are able to successfully swap, effectively
incrementing the value of `v`
}
```

Knowledge Check:

- race conditions are prevented by requiring that critical regions be protected by locks
  - true
- `test_and_set()` instruction is executed atomically
  - true
- an instruction that executes atomically
  - executes as a single, uninterruptible unit

OS & Programming Languages Approaches

Mutex Locks

Previous solutions are compicated and generally inaccessible to application programmers.

OS designers build software tools to solve critical section problem.

The simplest solution is the mutex lock.

> fun fact: mutex is a portmanteau of `mut`ual `ex`clusion, as it was created to solve that problem

Common Concurrency 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 *critical sections*. 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.

note that the mutex is an integer that alternates between 0 for locked and 1 to unlocked, this is effectively a boolean.

Protect a critical section by first `acquire()` a lock then `release()` the lock. Boolean variable indicating if `lock` is available or not.

`acquire()` and `release()` must be atomic. Usually implemented via hardware atomic instructions such as compare-and-swap.

```
while (true) {
  acquire lock

  critical section

  release lock

  remainder section
}
```

You `acquire()` the lock and no one else can touch it while you're in the critical section. Afterwards you can `release()` since you're done with the critical section.

The instructions look something like this in implementation.

```
acquire() {
  while (!available) { /* busy wait */}
  available = false;
}
release () {
  available = true;
}
```

- these two functions must be implemented atomically, typically through the use of something like `test_and_set()` and `compare_and_swap()`

cons:

- sol'n requires busy waiting
  - a process trying to enter enter its critical section will loop call `acquire()` over and over again to try and grab hold of it
- this lock is called a *spinlock*
  - if a lock is to be held for a `short duration`, one thread can "spin" on one processing core while another thread performs its critical section on another core.

short duration:

- how short is short?
- waiting on a lock requires 2 context switches
  - move the thread to the waiting state
    - storing away all of the process info (pcb, psw, etc.) introduces overhead
  - restore the waiting thread once the lock becomes available
- the general rule is to use a spinlock if the lock will be held for a duration of less than two context switches
- we don't know how long it will take for a critical section to take but we can get some good estimates

---

Knowledge check

- a mutex lock
  - is essentially a boolean variable
- what is the correct order of operations for protecting a critical section using mutex locks?
  - `acquire()` followed by `release()`
- busy waiting refers to the phenomenonthat while a process is in its critical section, any other prcess that tries to enter its critical section must loop continuously in the call to acquire the mutex lock
  - yes

```
Prof very briefly touched upon a notion of 2 different kinds of implementation of
semaphore and mutex.
Something about a classic implementation versus some other implementation.
```

Semaphores

One problem we had with the mutex was the busy waiting.

basic idea:

- $\geq$ 2 processes can work together using simple signals
- a process can be forced to stop at a specified place until it has received a specific signal
- any complex coordination can be done with an appropriate amount of signals
- our previously mentioned mutex is akin to a single flag that we can raise up and down.

Sempahores are a more sophisticated synchronization tool than mutex locks.

A semaphore `S` is an integer variable, that can only be accessed through 2 atomic operations:

1. `wait()` - decrements the semaphore value
2. `signal()` - increments the semaphore value

```
these each have other names.
wait() originally P() from Dutch proberen, "to test", semWait()
signal() originally V() from verhogen, "to increment", semSignal()

we may find these in other literature
```

Implementation

```
wait(S) {
  while (S <= 0) {/* busy wait */}
  S--;
}
signal(S) {
  S++;
}
```

There are 2 different kinds of semaphores:

1. counting sempahore
2. binary semaphore

Counting semaphores (aka general semaphore) is an integer value that can range over an unrestricted domain

Binary semaphores are an integer value that can only range between 0 and 1. This is a mutex lock.

We can implement a counting semaphore as a binary semaphore.

Good tool to solve synch problems.

- Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

- Consider P_1 and P_2 processes that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Initializing the semaphore to 0 will allow us to only allow P_2 to execute S_2 after the execution of S_1.

A semaphore must guarantee that no 2 processes can execute the `wait()` and `signal()` on the same semaphore at the same time

implementation becomes the critical sectin problem where the `wait` and `signal` code are placed in critical section.

Could now have busy waiting in critical section implementation.

Process can just suspend instead of waiting for the semaphore. Each semaphore has a waiting queue w/ 2 items

- value (of type integer)
- pointer to next record in the list

2 operations:

- block (aka sleep) - place the process invoking the operation on the appropriate waiting queue
- wakeup - remove one of the processes from the waiting queue and place in ready queue

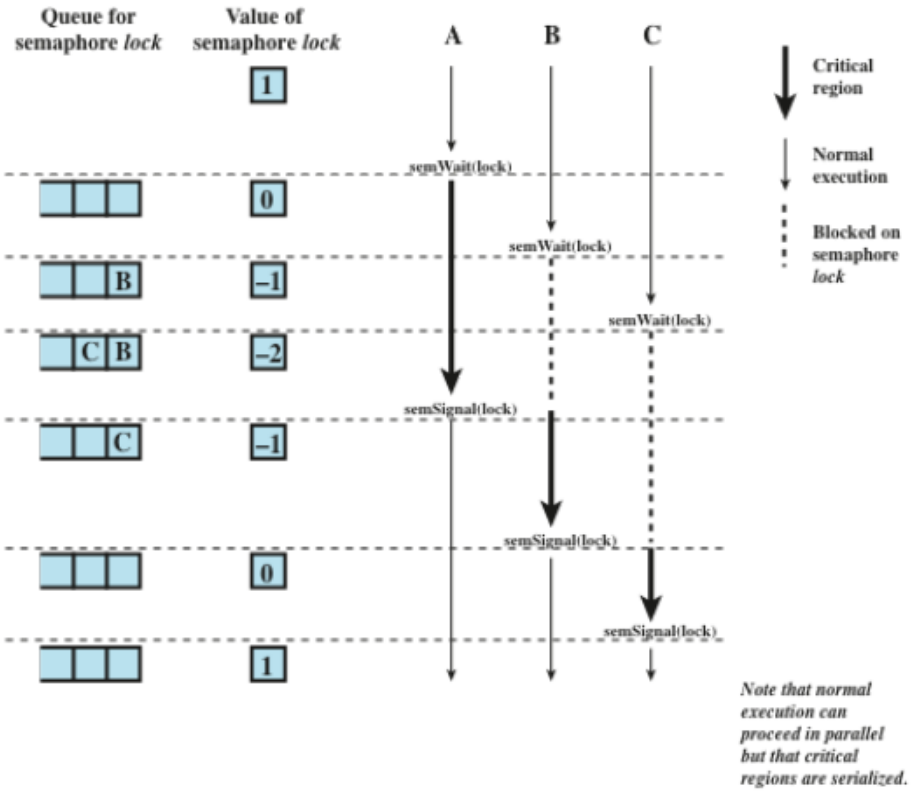
waiting queue:

```
typedef struct {
  int value;
  struct process *list;
} semaphore
```

No busy waiting implementation:

```
wait (semaphore *S) {
  S->value--;

  if (S->value < 0) {
    add this process to S->list;
    block;
  }
}

signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
    remove a process P from S->list;
    wakeup(P);
  }
}
```

Semaphores Usage Example



Processes Accessing Shared Resource Protected by a Semaphore

We're using the incrementing and decrementing to iterate through the queue.

A strong semaphore uses a queue of blocked processes, the process unblocked is the one in the queue for the longest time.

A weak semaphore uses a set of blocked processes. The processes unblocked is unpredictable. This can lead to process starvation as a process may remained blocked forever

Problems with Semaphores

- Incorrect use of semaphore operations:

```
signal(mutex);
        ...
        critical section
        ...
wait(mutex);
```

- or

```
wait(mutex);
        ...
        critical section
        ...
wait(mutex);
```

- Or omitting of **wait(mutex)** and/or **signal(mutex)**
- In these cases, either mutual exclusion is violated, or the process will permanently block.

---

knowledge check

- a counting semaphore
  - is essentially an integer variable
- ___ can be used to prevent busy waiting when implementing a semaphore
  - waiting queues
- mutex locks and binary sempahores are essentially the same thing
  - true

Monitors

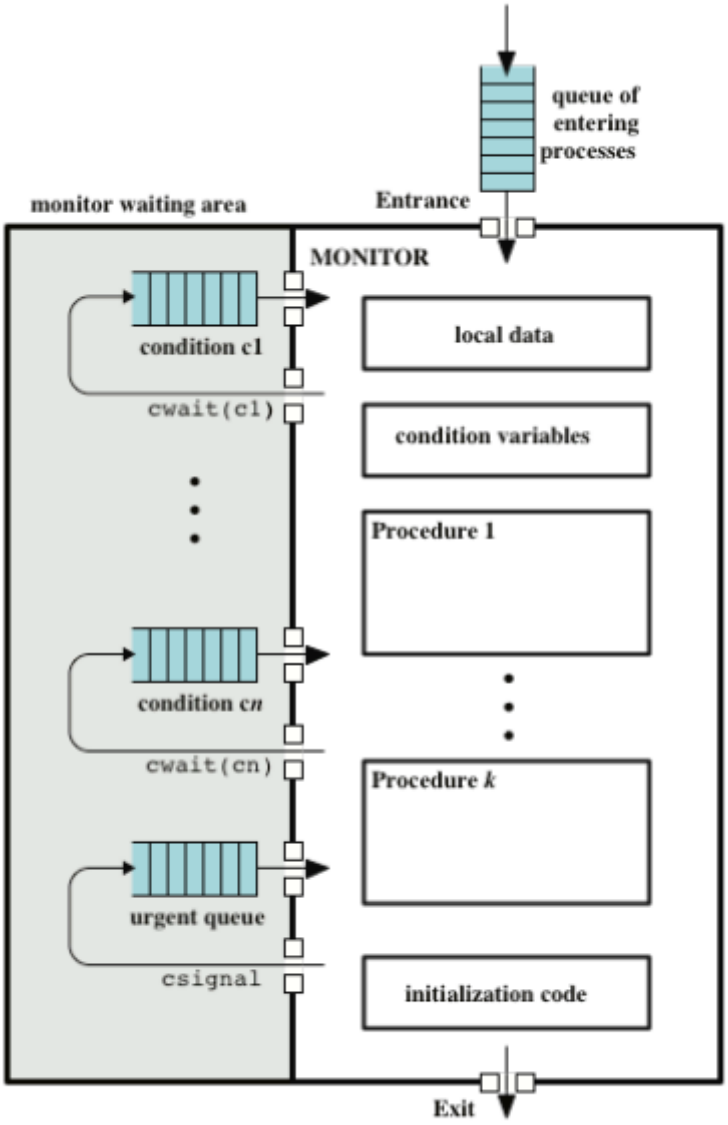
- High-level abstraction
  - easily accessible for the programmer
- used for process synchronization
- convenient and effective
- implemented in many programming languages including Java

implementation details?

- local data varialbes accessible only by the monitor's procedures and not by any external procedure
- a process enters the monitor by invoking one of its procedures
- only 1 process can be executing in the monitor at a time

```
The monitor achieves mutual exclusion by only allowing one process to run within it
at a time
```

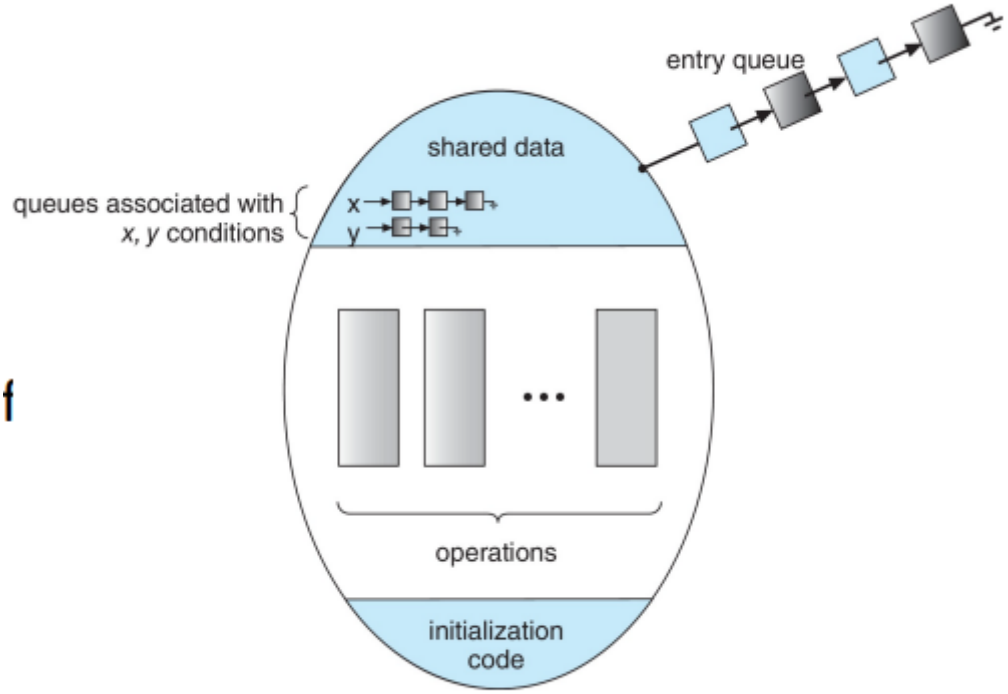
```
monitor monitor-name
{
        // shared variable
declarations
        function P1 (…) { …. }

        function P2 (…) { …. }

        function Pn (…) { …. }

   initialization code (…) { … }
}
```



Monitor with condition variables.

Condition Variables:

- we use condition variables to add more functions to the monitor
  - this is good for other synchronization schemes that need more functionality



f

Monitor with condition variables.

- `condition x, y;`
- two operations are allowed on a condition variable:
  - `x.wait()`
    - process that invokes it is susupended until `x.signal()`
  - `x.signal()`
    - if `x` invoked `x.wait()` previously then the process resumes
    - else, there is no effect on the variable

liveliness:

- a set of properties that a system must satisfy to ensure proesses make progress

liveliness and us:

- processes may have to wait indefinitely while trying to acquire a synchronization tool
- this violates the progress and bounded-waiting criteria that we talked about earlier
- and so this is an example of a liveliness failure

- **Deadlock** – 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	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- Consider if P_0 executes wait(S) and P_1 wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q)
- However, P_1 is waiting until P_0 execute signal(S).
- Since these signal() operations will never be executed, P_0 and P_1 are **deadlocked**.
- We also have something called **Livelock**!

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

---

knowledge check

- a ___ type presents a set of programmer-defined operatiosn that are provided mutual exclusion within it
  - monitor
- the local variables of a monitor can be accessed by only the local procedures
  - true
  - it is accessible only by the monitor's procedures which are the local procedures in this context
- monitors are a theoretical concept and are not practiced in modern programming languages
  - false
- deadlock-free solution eliminates the possibility of starvation
  - false
  - we can have livelocks and other forms of starvation
- another problem related to deadlocks is ___
  - spinlocks