操作系统原理

第七章：进程同步

洪明坚

重庆大学软件学院

February 19, 2016

目录

Outline

Background

Background

- Cooperating processes may share a block of memory via IPC facilities provided by the kernel.

Background

- Cooperating processes may share a block of memory via IPC facilities provided by the kernel.
  - Multiple threads within a process may share a piece of memory by using global variables.

Background

- Cooperating processes may share a block of memory via IPC facilities provided by the kernel.
  - Multiple threads within a process may share a piece of memory by using global variables.
- Concurrent access to shared data may result in data inconsistency.

Background

- Cooperating processes may share a block of memory via IPC facilities provided by the kernel.
  - Multiple threads within a process may share a piece of memory by using global variables.
- Concurrent access to shared data may result in data inconsistency.
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Background

- Cooperating processes may share a block of memory via IPC facilities provided by the kernel.
    - Multiple threads within a process may share a piece of memory by using global variables.
- Concurrent access to shared data may result in data inconsistency.
    - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Solution to *producer and consumer problem* (Chapter 4) allows at most *(BSIZE - 1)* items in buffer at the same time.

Background

- Cooperating processes may share a block of memory via IPC facilities provided by the kernel.
  - Multiple threads within a process may share a piece of memory by using global variables.
- Concurrent access to shared data may result in data inconsistency.
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Solution to *producer and consumer problem* (Chapter 4) allows at most *(BSIZE - 1)* items in buffer at the same time.
  - A solution, where all *BSIZE* buffers are used is NOT simple.

Background

- Cooperating processes may share a block of memory via IPC facilities provided by the kernel.
  - Multiple threads within a process may share a piece of memory by using global variables.
- Concurrent access to shared data may result in data inconsistency.
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Solution to *producer and consumer problem* (Chapter 4) allows at most *(BSIZE - 1)* items in buffer at the same time.
  - A solution, where all *BSIZE* buffers are used is NOT simple.
  - Suppose that we modify the code by adding a variable *counter*.

Producer and consumer problem

Producer and consumer problem

- Shared-memory bounded-buffer

Producer and consumer problem

- Shared-memory bounded-buffer

```
/* Shared variables */
#define BSIZE 10
struct item {
    ....
} buffer[BSIZE];
int in = 0, out = 0, counter = 0;
/*------------------------------------------------------------------*/
```

```
/*The producer loop*/
while(1) {
  /*produce an item*/
  while(counter == BSIZE)
    /*do nothing*/;
  buffer[in] = itemProduced;
  in = (in + 1) % BSIZE;
  counter++;
}
```

```
/*The consumer loop*/
while(1) {
  while(counter == 0)
    /*do nothing*/;
  itemConsumed = buffer[out];
  out = (out + 1) % BSIZE;
  counter--;
  /*consume the item*/
}
```

Problem of above solution (1/2)

Problem of above solution (1/2)

- When producer and consumer processes are executed *concurrently*, they may not function correctly.

Problem of above solution (1/2)

- When producer and consumer processes are executed *concurrently*, they may not function correctly.
  - We can show that the value of *counter* may be incorrect as follows.

Problem of above solution (1/2)

- When producer and consumer processes are executed *concurrently*, they may not function correctly.
    - We can show that the value of *counter* may be incorrect as follows.
- Note that the "$counter{+}{+}$" and "$counter{-}{-}$" may be implemented in machine language as

Problem of above solution (1/2)

- When producer and consumer processes are executed *concurrently*, they may not function correctly.
  - We can show that the value of *counter* may be incorrect as follows.
- Note that the "*counter*++" and "*counter*−−" may be implemented in machine language as

```
; counter++
register1 = counter           ; load the counter to a register
register1 = register1 + 1     ; add
counter = register1           ; write back

; counter--
register2 = counter           ; load the counter to a register
register2 = register2 - 1     ; subtract
counter = register2           ; write back
```

Problem of above solution (2/2)

Problem of above solution (2/2)

- It's important to note that the above two instruction sequences can be *interleaved* because of interrupts or scheduling.

Problem of above solution (2/2)

- It's important to note that the above two instruction sequences can be *interleaved* because of interrupts or scheduling.
- For example, assume *counter* is initially 5,

Problem of above solution (2/2)

- It's important to note that the above two instruction sequences can be *interleaved* because of interrupts or scheduling.
- For example, assume *counter* is initially 5,

Time	Producer	Consumer	result

Problem of above solution (2/2)

- It's important to note that the above two instruction sequences can be *interleaved* because of interrupts or scheduling.
- For example, assume *counter* is initially 5,

Time	Producer	Consumer	result
T_0	register1=*counter*		{register1=5}

Problem of above solution (2/2)

- It's important to note that the above two instruction sequences can be *interleaved* because of interrupts or scheduling.
- For example, assume *counter* is initially 5,

Time	Producer	Consumer	result
T_0	register1=*counter*		{register1=5}
T_1	register1=register1+1		{register1=6}

Problem of above solution (2/2)

- It's important to note that the above two instruction sequences can be *interleaved* because of interrupts or scheduling.
- For example, assume *counter* is initially 5,

Time	Producer	Consumer	result
T_0	register1=*counter*		{register1=5}
T_1	register1=register1+1		{register1=6}
T_2		register2=counter	{register2=5}

Problem of above solution (2/2)

- It's important to note that the above two instruction sequences can be *interleaved* because of interrupts or scheduling.
- For example, assume *counter* is initially 5,

Time	Producer	Consumer	result
T_0	register1=*counter*		{register1=5}
T_1	register1=register1+1		{register1=6}
T_2		register2=counter	{register2=5}
T_3		register2=register2-1	{register2=4}

Problem of above solution (2/2)

- It's important to note that the above two instruction sequences can be *interleaved* because of interrupts or scheduling.
- For example, assume *counter* is initially 5,

Time	Producer	Consumer	result
T_0	register1=*counter*		{register1=5}
T_1	register1=register1+1		{register1=6}
T_2		register2=counter	{register2=5}
T_3		register2=register2-1	{register2=4}
T_4	*counter*=register1		{*counter*=6}

Problem of above solution (2/2)

- It's important to note that the above two instruction sequences can be *interleaved* because of interrupts or scheduling.
- For example, assume *counter* is initially 5,

Time	Producer	Consumer	result
T_0	register1=*counter*		{register1=5}
T_1	register1=register1+1		{register1=6}
T_2		register2=counter	{register2=5}
T_3		register2=register2-1	{register2=4}
T_4	*counter*=register1		{*counter*=6}
T_5		*counter*=register2	{*counter*=4}

Problem of above solution (2/2)

- It's important to note that the above two instruction sequences can be *interleaved* because of interrupts or scheduling.
- For example, assume *counter* is initially 5,

Time	Producer	Consumer	result
T_0	register1=*counter*		{register1=5}
T_1	register1=register1+1		{register1=6}
T_2		register2=counter	{register2=5}
T_3		register2=register2-1	{register2=4}
T_4	*counter*=register1		{*counter*=6}
T_5		*counter*=register2	{*counter*=4}

- The value of *counter* may be either 4 or 6, while the correct result should be 5.

Questions

- Any questions?

Race condition (1/2)

Race condition (1/2)

- *Race condition is a situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.*

Race condition (1/2)

- *Race condition is a situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.*
- How do we avoid race condition?

Race condition (1/2)

- *Race condition is a situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.*
- How do we avoid race condition?
  - Find some way to prohibit more than one process from reading and writing the shared data concurrently.

Race condition (1/2)

- *Race condition is a situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place.*
- How do we avoid race condition?
  - Find some way to prohibit more than one process from reading and writing the shared data concurrently.
  - That is, the access must be *serialized* even if the processes attempt concurrent access.

Race condition (2/2)

Race condition (2/2)

- **Critical section**

- **Critical section**
  - a piece of code where the shared resource is accessed.

Race condition (2/2)

- **Critical section**
  - a piece of code where the shared resource is accessed.
- A solution of race condition must satisfy the following 4 requirements:

Race condition (2/2)

- **Critical section**
  - a piece of code where the shared resource is accessed.
- A solution of race condition must satisfy the following 4 requirements:

  1. **Mutual exclusion**: No two process may be simultaneously inside their critical section;

Race condition (2/2)

- **Critical section**
  - a piece of code where the shared resource is accessed.
- A solution of race condition must satisfy the following 4 requirements:

1. **Mutual exclusion**: No two process may be simultaneously inside their critical section;
2. **Progress**: No process running outside its critical section may block other processes trying to enter its critical section;

Race condition (2/2)

- **Critical section**
  - a piece of code where the shared resource is accessed.
- A solution of race condition must satisfy the following 4 requirements:

1. **Mutual exclusion**: No two process may be simultaneously inside their critical section;
2. **Progress**: No process running outside its critical section may block other processes trying to enter its critical section;
3. **Bounded waiting**: No process should have to wait forever to enter its critical section;

- **Critical section**
  - a piece of code where the shared resource is accessed.
- A solution of race condition must satisfy the following 4 requirements:

1. **Mutual exclusion**: No two process may be simultaneously inside their critical section;
2. **Progress**: No process running outside its critical section may block other processes trying to enter its critical section;
3. **Bounded waiting**: No process should have to wait forever to enter its critical section;
4. **Speed**: No assumptions may be made about speeds or the number of the CPUs.

Critical section

Critical section

- A protocol must be designed to be used by the processes to enter and leave critical section.

Critical section

- A protocol must be designed to be used by the processes to enter and leave critical section.
  - Each process must request permission to enter its critical section.

Critical section

- A protocol must be designed to be used by the processes to enter and leave critical section.
    - Each process must request permission to enter its critical section.

```
do {
  <entry section>

  CRITICAL SECTION

  <exit section>

} while (1);
```

Critical section

- A protocol must be designed to be used by the processes to enter and leave critical section.
  - Each process must request permission to enter its critical section.

```
do {
  <entry section>

  CRITICAL SECTION

  <exit section>

} while (1);
```

- In the following slides, we will try to construct several "entry section" and "exit section" to solve the race condition.

Questions

- Any questions?

Outline

First try

First try

- Assume there are only two cooperating processes: P_i and P_j, where i+j=1.

First try

- Assume there are only two cooperating processes: P_i and P_j, where i+j=1.

```
int turn = 0; // or 1
do {
  while(turn != i);   // entry section

  CRITICAL SECTION

  turn = j;           // exit section

} while (1);
```

- Does it satisfy all 4 requirements for a solution?

First try

- Assume there are only two cooperating processes: P_i and P_j, where i+j=1.

```
int turn = 0; // or 1
do {
  while(turn != i);   // entry section

  CRITICAL SECTION

  turn = j;           // exit section

} while (1);
```

- Does it satisfy all 4 requirements for a solution?
  - No, it breaks the **progress** requirement.

Second try

Second try

```
bool flag[2] = {false, false};
do {
  // entry section
  flag[i] = true;
  while(flag[j]);

  CRITICAL SECTION

  flag[i] = false; // exit section

} while (1);
```

- Does it satisfy all 4 requirements for a solution?

Second try

```
bool flag[2] = {false, false};
do {
  // entry section
  flag[i] = true;
  while(flag[j]);

  CRITICAL SECTION

  flag[i] = false; // exit section

} while (1);
```

- Does it satisfy all 4 requirements for a solution?
  - No, it breaks the **progress** requirement too.

Third try

Third try

```
int turn = 0; // or 1
bool flag[2] = {false, false};
do {
  // entry section
  flag[i] = true;
  turn = j;
  while(flag[j] && turn == j);

  CRITICAL SECTION

  flag[i] = false; // exit section

} while (1);
```

- Does it satisfy all 4 requirements for a solution?

Third try

```
int turn = 0; // or 1
bool flag[2] = {false, false};
do {
  // entry section
  flag[i] = true;
  turn = j;
  while(flag[j] && turn == j);

  CRITICAL SECTION

  flag[i] = false; // exit section

} while (1);
```

- Does it satisfy all 4 requirements for a solution?
  - Yes, it's a correct solution and is known as *Peterson's algorithm*.

Bakery algorithm

Bakery algorithm

- Peterson's algorithm solves the problem for two processes, while *Bakery algorithm* solves it for multiple processes.

Bakery algorithm

- Peterson's algorithm solves the problem for two processes, while *Bakery algorithm* solves it for multiple processes.

```
bool choosing[n] = {false, ..., false};
int number[n] = {0, ..., 0};
do {
  // entry section
  choosing[i] = true;
  number[i] = max(number[0], ..., number[n-1])+1;
  choosing[i] = false;
  for(j=0; j < n; j++) {
    while(choosing[j]);
    while((number[j]!=0)&&(number[j],j)<(number[i],i));
  }

  CRITICAL SECTION

  number[i] = 0; // exit section

} while (1);
```

Questions

- Any questions?

Synchronization hardware

Synchronization hardware

- Several low-level hardware features may be used to solve the race condition.

Synchronization hardware

- Several low-level hardware features may be used to solve the race condition.
  1. Disabling interrupts

Synchronization hardware

- Several low-level hardware features may be used to solve the race condition.
  1. Disabling interrupts
  2. Special instructions

Synchronization hardware

- Several low-level hardware features may be used to solve the race condition.
  1. Disabling interrupts
  2. Special instructions
     - TSL (Test and Set Lock)

Synchronization hardware

- Several low-level hardware features may be used to solve the race condition.
  1. Disabling interrupts
  2. Special instructions
     - TSL (Test and Set Lock)
     - SWAP

Disabling interrupts

Disabling interrupts

- The CPU is only switched from process to process as a result of clock or other interrupts.

Disabling interrupts

- The CPU is only switched from process to process as a result of clock or other interrupts.
    - Once a process has disabled interrupts, it can access the shared memory without fear that any other process will intervene.

Disabling interrupts

- The CPU is only switched from process to process as a result of clock or other interrupts.
  - Once a process has disabled interrupts, it can access the shared memory without fear that any other process will intervene.
- Disadvantages

Disabling interrupts

- The CPU is only switched from process to process as a result of clock or other interrupts.
  - Once a process has disabled interrupts, it can access the shared memory without fear that any other process will intervene.
- Disadvantages
  1. User processes should NOT be able to disable the interrupts;

Disabling interrupts

- The CPU is only switched from process to process as a result of clock or other interrupts.
  - Once a process has disabled interrupts, it can access the shared memory without fear that any other process will intervene.
- Disadvantages
  1. User processes should NOT be able to disable the interrupts;
  2. It's not feasible in a multiprocessor system.

TSL and SWAP (1/3)

TSL and SWAP (1/3)

- The TSL and SWAP instructions have the following functionalities, respectively:

TSL and SWAP (1/3)

- The TSL and SWAP instructions have the following functionalities, respectively:

```
bool TSL(bool &target)
{
  bool rv = target;
  target = true;
  return rv;
}
```

```
void SWAP(bool &a, bool &b)
{
  bool temp = a;
  a = b;
  b= temp;
}
```

TSL and SWAP (1/3)

- The TSL and SWAP instructions have the following functionalities, respectively:

```
bool TSL(bool &target)
{
  bool rv = target;
  target = true;
  return rv;
}
```

```
void SWAP(bool &a, bool &b)
{
  bool temp = a;
  a = b;
  b= temp;
}
```

- Bear in mind that TSL and SWAP are executed *atomically*, that is, as one uninterruptible unit.

TSL and SWAP (2/3)

TSL and SWAP (2/3)

- Use TSL or SWAP to TRY to solve race condition

TSL and SWAP (2/3)

- Use TSL or SWAP to TRY to solve race condition

```
bool lock = false;
do {
  // entry section
  while(TSL(lock))
    ;

  CRITICAL SECTION

  // exit section
  lock = false;

} while (1);
```

```
bool lock=false;
do {
  // entry section
  bool key = true;
  while(key == true)
    SWAP(lock, key);

  CRITICAL SECTION

  // exit section
  lock = false;

} while (1);
```

TSL and SWAP (2/3)

- Use TSL or SWAP to TRY to solve race condition

```
bool lock = false;
do {
  // entry section
  while(TSL(lock))
    ;

  CRITICAL SECTION

  // exit section
  lock = false;

} while (1);
```

```
bool lock=false;
do {
  // entry section
  bool key = true;
  while(key == true)
    SWAP(lock, key);

  CRITICAL SECTION

  // exit section
  lock = false;

} while (1);
```

- But these two algorithms do NOT satisfy the **bounded-waiting** requirement.

TSL and SWAP (3/3)

TSL and SWAP (3/3)

- A correct solution using TSL

TSL and SWAP (3/3)

- A correct solution using TSL

```
bool lock = false, waiting[n] = {false, ..., false};
do {
  // entry section
  waiting[i] = true;  bool key = true;
  while(waiting[i] && key)
    key = TSL(lock);
  waiting[i] = false;

  CRITICAL SECTION

  // exit section
  int j = (i + 1) % n;
  while((j != i) && !waiting[j])
    j = (j+1) % n;
  if(j == i)      lock = false;
  else            waiting[j] = false;

} while (1);
```

Conclusion

Conclusion

- The above solutions have a common disadvantage: *busy waiting*.

Conclusion

- The above solutions have a common disadvantage: *busy waiting*.
    - While a process is in its critical section, any other process tries to enter its critical section must loop continuously in the entry code.

Conclusion

- The above solutions have a common disadvantage: *busy waiting*.
  - While a process is in its critical section, any other process tries to enter its critical section must loop continuously in the entry code.
  - Busy waiting wastes CPU cycles that some other process might be able to use productively.

Conclusion

- The above solutions have a common disadvantage: *busy waiting*.
  - While a process is in its critical section, any other process tries to enter its critical section must loop continuously in the entry code.
  - Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of solution is also called a *spinlock*.

Conclusion

- The above solutions have a common disadvantage: *busy waiting*.
  - While a process is in its critical section, any other process tries to enter its critical section must loop continuously in the entry code.
  - Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of solution is also called a *spinlock*.
  - Because the process “spins” while waiting on a lock.

Conclusion

- The above solutions have a common disadvantage: *busy waiting*.
    - While a process is in its critical section, any other process tries to enter its critical section must loop continuously in the entry code.
    - Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of solution is also called a *spinlock*.
    - Because the process “spins” while waiting on a lock.
    - The spinlocks are **only** useful in multiprocessor systems.

Conclusion

- The above solutions have a common disadvantage: *busy waiting*.
  - While a process is in its critical section, any other process tries to enter its critical section must loop continuously in the entry code.
  - Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of solution is also called a *spinlock*.
  - Because the process "spins" while waiting on a lock.
  - The spinlocks are **only** useful in multiprocessor systems.
    - Because no context switch is required if the locks are expected to be held for short times.

Questions

- Any questions?

Outline

Semaphore

Semaphore

- The above solutions to race condition are not easy to generalize to more complex problems.

Semaphore

- The above solutions to race condition are not easy to generalize to more complex problems.
- To overcome this difficulty, we can use a synchronization tool called *semaphore*.

Semaphore

- The above solutions to race condition are not easy to generalize to more complex problems.
- To overcome this difficulty, we can use a synchronization tool called *semaphore*.
  - It's invented by *Edsger Dijkstra* and first used in the *THE* operating system.

Semaphore

- The above solutions to race condition are not easy to generalize to more complex problems.
- To overcome this difficulty, we can use a synchronization tool called *semaphore*.
    - It's invented by *Edsger Dijkstra* and first used in the *THE* operating system.
- What's the semaphore?

Semaphore

- The above solutions to race condition are not easy to generalize to more complex problems.
- To overcome this difficulty, we can use a synchronization tool called *semaphore*.
  - It's invented by *Edsger Dijkstra* and first used in the *THE* operating system.
- What's the semaphore?
  - A semaphore S is an integer variable that, apart from initialization, is accessed **only** through two standard **atomic** operations: $P(down)$ and $V(up)$.

Implementation (1/2)

Implementation (1/2)

```
typedef struct {
  int value;

  /*processes blocked by this semaphore*/
  struct PCB *L;
} semaphore;
```

```
void P(semaphore *S) {
 S->value--;
 if(S->value < 0) {
  Add (curproc) to S->L;

  (curproc)->state=WAITING;
  scheduler();
 }
}
```

```
void V(semaphore *S) {
  S->value++;
  if(S->value <= 0) {
    Remove a proc (A) from S->L;

    (A)->state=READY; //Wakeup (A)

  }
}
```

Implementation (2/2)

Implementation (2/2)

- The magnitude of the “value” is the number of resources available (>0) or of processes waiting on that semaphore(<0).

Implementation (2/2)

- The magnitude of the "value" is the number of resources available (>0) or of processes waiting on that semaphore(<0).
- The P and V must be executed **atomically**.

Implementation (2/2)

- The magnitude of the “value” is the number of resources available (>0) or of processes waiting on that semaphore(<0).
- The P and V must be executed **atomically**.
  - In either of two ways:

Implementation (2/2)

- The magnitude of the "value" is the number of resources available ($>$0) or of processes waiting on that semaphore($<$0).
- The P and V must be executed **atomically**.
  - In either of two ways:
    1. Disable interrupts in uni-processor systems or

Implementation (2/2)

- The magnitude of the "value" is the number of resources available (>0) or of processes waiting on that semaphore(<0).
- The *P* and *V* must be executed **atomically**.
    - In either of two ways:
        1. Disable interrupts in uni-processor systems or
        2. Spinlocks in multi-processor systems.

Application programming interface

Application programming interface

- Win32

Application programming interface

- Win32
  - CreateSemaphore/CloseHandle

Application programming interface

- Win32
  - CreateSemaphore/CloseHandle
  - WaitForSingleObject/ReleaseSemaphore

Application programming interface

- Win32
  - CreateSemaphore/CloseHandle
  - WaitForSingleObject/ReleaseSemaphore
- POSIX

Application programming interface

- Win32
  - CreateSemaphore/CloseHandle
  - WaitForSingleObject/ReleaseSemaphore
- POSIX
  - sem_init/sem_destroy

Application programming interface

- Win32
  - CreateSemaphore/CloseHandle
  - WaitForSingleObject/ReleaseSemaphore
- POSIX
  - sem_init/sem_destroy
  - sem_wait/sem_post

Questions

- Any questions?

Classic problems of synchronization

Classic problems of synchronization

- The producer and consumer problem

Classic problems of synchronization

- The producer and consumer problem
  - Explained in the chapter “Process Management”.

Classic problems of synchronization

- The producer and consumer problem
  - Explained in the chapter “Process Management”.
- The readers-writers problem

Classic problems of synchronization

- The producer and consumer problem
  - Explained in the chapter “Process Management”.
- The readers-writers problem
  - A data object is to be shared among several concurrent processes.

Classic problems of synchronization

- The producer and consumer problem
  - Explained in the chapter “Process Management”.
- The readers-writers problem
  - A data object is to be shared among several concurrent processes.
    - Some of them (*readers*) may want **only** to read the data object,

Classic problems of synchronization

- The producer and consumer problem
  - Explained in the chapter "Process Management".
- The readers-writers problem
  - A data object is to be shared among several concurrent processes.
    - Some of them (*readers*) may want **only** to read the data object, while others (*writers*) may want to update the data object.

Classic problems of synchronization

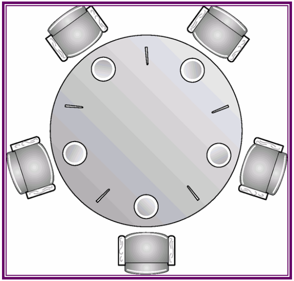
- The producer and consumer problem
  - Explained in the chapter "Process Management".
- The readers-writers problem
  - A data object is to be shared among several concurrent processes.
    - Some of them (*readers*) may want **only** to read the data object, while others (*writers*) may want to update the data object.
  - Multiple readers may read the data object simultaneously, but the writers must have exclusive access to the data object.

Classic problems of synchronization

- The producer and consumer problem
  - Explained in the chapter "Process Management".
- The readers-writers problem
  - A data object is to be shared among several concurrent processes.
    - Some of them (*readers*) may want **only** to read the data object, while others (*writers*) may want to update the data object.
  - Multiple readers may read the data object simultaneously, but the writers must have exclusive access to the data object.
- The dining-philosophers problem

Classic problems of synchronization

- The producer and consumer problem
  - Explained in the chapter “Process Management”.
- The readers-writers problem
  - A data object is to be shared among several concurrent processes.
    - Some of them (*readers*) may want **only** to read the data object, while others (*writers*) may want to update the data object.
  - Multiple readers may read the data object simultaneously, but the writers must have exclusive access to the data object.
- The dining-philosophers problem



The producer and consumer problem (1/2)

The producer and consumer problem (1/2)

- A semaphore *mutex* is used to protect the critical section when accessing the buffer.

The producer and consumer problem (1/2)

- A semaphore *mutex* is used to protect the critical section when accessing the buffer.
  - It's initialized to the value 1.

The producer and consumer problem (1/2)

- A semaphore *mutex* is used to protect the critical section when accessing the buffer.
  - It's initialized to the value 1.
- The semaphores *empty* and *full* synchronize the producer and consumer.

The producer and consumer problem (1/2)

- A semaphore *mutex* is used to protect the critical section when accessing the buffer.
  - It's initialized to the value 1.
- The semaphores *empty* and *full* synchronize the producer and consumer.
  - *empty* is initialized to the value *BSIZE*;

The producer and consumer problem (1/2)

- A semaphore *mutex* is used to protect the critical section when accessing the buffer.
  - It's initialized to the value 1.
- The semaphores *empty* and *full* synchronize the producer and consumer.
  - *empty* is initialized to the value *BSIZE*;
  - *full* is initialized to the value 0.

The producer and consumer problem (2/2)

The producer and consumer problem (2/2)

```
/*The producer loop*/
do {
    ...
    produce an item;
    ...
   P(&empty);
   P(&mutex);
    ...
   add the item to buffer;
    ...
   V(&mutex);
   V(&full);
} while(1);
```

The producer and consumer problem (2/2)

```
/*The producer loop*/
do {
   ...
   produce an item;
   ...
   P(&empty);
   P(&mutex);
   ...
   add the item to buffer;
   ...
   V(&mutex);
   V(&full);
} while(1);
```

```
/*The consumer loop*/
do {
   P(&full);
   P(&mutex);
   ...
   remove an item from buffer;
   ...
   V(&mutex);
   V(&empty);
   ...
   consume the item;
   ...
} while(1);
```

Questions

- Any questions?

The readers-writers problem (1/2)

The readers-writers problem (1/2)

- A semaphore *wrt* is used to protect the shared data object.

The readers-writers problem (1/2)

- A semaphore *wrt* is used to protect the shared data object.
    - It's initialized to the value 1.

The readers-writers problem (1/2)

- A semaphore *wrt* is used to protect the shared data object.
    - It's initialized to the value 1.
- An integer *readcount* count the number of readers which is busy reading.

The readers-writers problem (1/2)

- A semaphore *wrt* is used to protect the shared data object.
    - It's initialized to the value 1.
- An integer *readcount* count the number of readers which is busy reading.
    - *readcount* is initialized to the value 0;

The readers-writers problem (1/2)

- A semaphore *wrt* is used to protect the shared data object.
  - It's initialized to the value 1.
- An integer *readcount* count the number of readers which is busy reading.
  - *readcount* is initialized to the value 0;
- Another semaphore *mutex* is used to protect the *readcount*.

The readers-writers problem (1/2)

- A semaphore *wrt* is used to protect the shared data object.
  - It's initialized to the value 1.
- An integer *readcount* count the number of readers which is busy reading.
  - *readcount* is initialized to the value 0;
- Another semaphore *mutex* is used to protect the *readcount*.
  - *mutex* is initialized to the value 1.

The readers-writers problem (2/2)

The readers-writers problem (2/2)

```
/*The writer loop*/
do {
    P(&wrt);
     ...
     writing is performed
     ...
    V(&wrt);
} while(1);
```

The readers-writers problem (2/2)

```
/*The writer loop*/
do {
    P(&wrt);
    ...
    writing is performed
    ...
    V(&wrt);
} while(1);
```

```
/*The reader loop*/
do {
    P(&mutex);
    readcount++;
    if(readcount == 1)
      P(&wrt);
    V(&mutex);
    ...
    reading is performed
    ...
    P(&mutex);
    readcount--;
    if(readcount == 0)
      V(&wrt);
    V(&mutex);
} while(1);
```

Questions

- Any questions?

The dining-philosopher problem (1/3)

The dining-philosopher problem (1/3)

```
semaphore chopstick[5] = {1, ..., 1};
/*philosopher i*/
do {
    ...
    thinking
    ...
   P(&chopstick[i]);
   P(&chopstick[(i + 1) % 5]);
    ...
    eating
    ...
   V(&chopstick[(i + 1) % 5]);
   V(&chopstick[i]);
} while(1);
```

The dining-philosopher problem (1/3)

```
semaphore chopstick[5] = {1, ..., 1};
/*philosopher i*/
do {
    ...
    thinking
    ...
   P(&chopstick[i]);
   P(&chopstick[(i + 1) % 5]);
    ...
    eating
    ...
   V(&chopstick[(i + 1) % 5]);
   V(&chopstick[i]);
} while(1);
```

- Is this a correct solution?

The dining-philosopher problem (1/3)

```
semaphore chopstick[5] = {1, ..., 1};
/*philosopher i*/
do {
   ...
   thinking
   ...
  P(&chopstick[i]);
  P(&chopstick[(i + 1) % 5]);
   ...
   eating
   ...
  V(&chopstick[(i + 1) % 5]);
  V(&chopstick[i]);
} while(1);
```

- Is this a correct solution?
  - No.

The dining-philosophers problem (2/3)

The dining-philosophers problem (2/3)

```
#define N          5
#define LEFT       ((i-1) % N)
#define RIGHT      ((i+1) % N)

enum {THINKING, HUNGRY, EATING} state[N];
semaphore mutex = 1; /*used to protect the 'state'*/

semaphore s[N] = {0, ..., 0};

               void philosopher(int i)
               {
                 while(1) {
                   think();
                   take_forks(i);
                   eat();
                   put_forks(i);
                 }
               }
```

The dining-philosophers problem (3/3)

The dining-philosophers problem (3/3)

```c
void test(int i)
{
   if(state[i] == HUNGRY &&
      state[LEFT] != EATING &&
      state[RIGHT] != EATING) {
     state[i] = EATING;
     V(&s[i]);
   }
}
```

The dining-philosophers problem (3/3)

```
void test(int i)
{
  if(state[i] == HUNGRY &&
     state[LEFT] != EATING &&
     state[RIGHT] != EATING) {
    state[i] = EATING;
    V(&s[i]);
  }
}
```

```
void take_forks(int i)
{
  P(&mutex);
  state[i] = HUNGRY;
  test(i);
  V(&mutex);
  P(&s[i]);
}
```

```
void put_forks(int i)
{
  P(&mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  V(&mutex);
}
```

Questions

- Any questions?

Binary semaphore

Binary semaphore

- The semaphore construct described in the previous slides is commonly known as a *counting semaphore*.

Binary semaphore

- The semaphore construct described in the previous slides is commonly known as a *counting semaphore*.
  - Since its value can range over an unrestricted domain.

Binary semaphore

- The semaphore construct described in the previous slides is commonly known as a *counting semaphore*.
    - Since its value can range over an unrestricted domain.
- A *binary semaphore* is a semaphore with an integer value range only between 0 and 1.

Binary semaphore

- The semaphore construct described in the previous slides is commonly known as a *counting semaphore*.
  - Since its value can range over an unrestricted domain.
- A *binary semaphore* is a semaphore with an integer value range only between 0 and 1.
  - It can be simpler to implement than a counting semaphore on some hardware architectures.

Binary semaphore

- The semaphore construct described in the previous slides is commonly known as a *counting semaphore*.
  - Since its value can range over an unrestricted domain.
- A *binary semaphore* is a semaphore with an integer value range only between 0 and 1.
  - It can be simpler to implement than a counting semaphore on some hardware architectures.
- A counting semaphore can be implemented using binary semaphores.

Implementation of binary semaphore

Implementation of binary semaphore

```
typedef struct {
  bool flag;

  /*processes blocked by this binary-semaphore*/
  struct PCB *L;
} binary-semaphore;
```

Implementation of binary semaphore

```
typedef struct {
  bool flag;

  /*processes blocked by this binary-semaphore*/
  struct PCB *L;
} binary-semaphore;
```

```
void bP(binary-semaphore *bS) {
  if(bS->flag == true)
    bS->flag = false;
  else {
    Add (curproc) to bS->L;

    (curproc)->state=WAITING;
    scheduler();
  }
}
```

Implementation of binary semaphore

```
typedef struct {
  bool flag;

  /*processes blocked by this binary-semaphore*/
  struct PCB *L;
} binary-semaphore;
```

```
void bP(binary-semaphore *bS) {
  if(bS->flag == true)
    bS->flag = false;
  else {
    Add (curproc) to bS->L;

    (curproc)->state=WAITING;
    scheduler();
  }
}
```

```
void bV(binary-semaphore *bS) {
  if(bS->L is empty)
    bS->flag = true;
  else {
    Remove a proc (A) from bS->L;

    (A)->state=READY; //Wakeup (A)
  }
}
```

Implementation of binary semaphore

```
typedef struct {
  bool flag;

  /*processes blocked by this binary-semaphore*/
  struct PCB *L;
} binary-semaphore;
```

```
void bP(binary-semaphore *bS) {
  if(bS->flag == true)
    bS->flag = false;
  else {
    Add (curproc) to bS->L;

    (curproc)->state=WAITING;
    scheduler();
  }
}
```

```
void bV(binary-semaphore *bS) {
  if(bS->L is empty)
    bS->flag = true;
  else {
    Remove a proc (A) from bS->L;

    (A)->state=READY; //Wakeup (A)
  }
}
```

- The *bP* and *bV* must be executed **atomically**.

Implement counting semaphore using binary semaphore

Implement counting semaphore using binary semaphore

```
typedef struct {
  int value;

  binary-semaphore bS1 = true,   /*protect 'value'*/
                   bS2 = false;  /*synchronization*/
} semaphore;
```

Implement counting semaphore using binary semaphore

```
typedef struct {
  int value;

  binary-semaphore bS1 = true,   /*protect 'value'*/
                   bS2 = false;  /*synchronization*/
} semaphore;
```

```
void P(semaphore *S)
{
  bP(&S->bS1);
  S->value--;
  if(S->value < 0) {
    bV(&S->bS1);
    bP(&S->bS2);
  } else
    bV(&S->bS1);
}
```

Implement counting semaphore using binary semaphore

```c
typedef struct {
  int value;

  binary-semaphore bS1 = true,  /*protect 'value'*/
                   bS2 = false; /*synchronization*/
} semaphore;
```

```c
void P(semaphore *S)
{
  bP(&S->bS1);
  S->value--;
  if(S->value < 0) {
    bV(&S->bS1);
    bP(&S->bS2);
  } else
    bV(&S->bS1);
}
```

```c
void V(semaphore *S)
{
  bP(&S->bS1);
  S->value++;
  if(S->value <= 0)
    bV(&S->bS2);
  bV(&S->bS1);
}
```

Implement counting semaphore using binary semaphore

```
typedef struct {
  int value;

  binary-semaphore bS1 = true,   /*protect 'value'*/
                   bS2 = false;  /*synchronization*/
} semaphore;
```

```
void P(semaphore *S)
{
  bP(&S->bS1);
  S->value--;
  if(S->value < 0) {
    bV(&S->bS1);
    bP(&S->bS2);
  } else
    bV(&S->bS1);
}
```

```
void V(semaphore *S)
{
  bP(&S->bS1);
  S->value++;
  if(S->value <= 0)
    bV(&S->bS2);
  bV(&S->bS1);
}
```

- This implementation of P and V may **not** be executed **atomically**.

Questions

- Any questions?

Deadlock and starvation (1/2)

Deadlock and starvation (1/2)

- Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect use can still result in hard-to-detect errors.

Deadlock and starvation (1/2)

- Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect use can still result in hard-to-detect errors.
    - For example, suppose that two *P*s in the producer loop were reversed in order. That is,

Deadlock and starvation (1/2)

- Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect use can still result in hard-to-detect errors.
  - For example, suppose that two *P*s in the producer loop were reversed in order. That is,

```
do { /*producer*/
   ...
   produce an item;
   ...
   P(&mutex); /*XXX*/
   P(&empty); /*XXX*/
   ...
   add the item to buffer;
   ...
   V(&mutex);
   V(&full);
} while(1);
```

Deadlock and starvation (1/2)

- Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect use can still result in hard-to-detect errors.
  - For example, suppose that two Ps in the producer loop were reversed in order. That is,

```
do { /*producer*/
   ...
   produce an item;
   ...
   P(&mutex); /*XXX*/
   P(&empty); /*XXX*/
   ...
   add the item to buffer;
   ...
   V(&mutex);
   V(&full);
} while(1);
```

```
do { /*consumer*/
   P(&full);
   P(&mutex);
   ...
   remove an item from buffer;
   ...
   V(&mutex);
   V(&empty);
   ...
   consume the item;
   ...
} while(1);
```

Deadlock and starvation (2/2)

Deadlock and starvation (2/2)

- So, what's a deadlock?

Deadlock and starvation (2/2)

- So, what's a deadlock?
  - *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*

Deadlock and starvation (2/2)

- So, what's a deadlock?
  - *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*
  - We will cover the deadlock in the next chapter.

Deadlock and starvation (2/2)

- So, what's a deadlock?
  - *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*
  - We will cover the deadlock in the next chapter.
- Another problem related to deadlocks is *starvation* or *indefinite blocking*.

Deadlock and starvation (2/2)

- So, what's a deadlock?
  - *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*
  - We will cover the deadlock in the next chapter.
- Another problem related to deadlocks is *starvation* or *indefinite blocking*.
  - A situation where processes wait indefinitely within the semaphore.

Deadlock and starvation (2/2)

- So, what's a deadlock?
  - *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*
  - We will cover the deadlock in the next chapter.
- Another problem related to deadlocks is *starvation* or *indefinite blocking*.
  - A situation where processes wait indefinitely within the semaphore.
  - For example, the previous solution to the 'readers-writers' problem may result in starvation.

Deadlock and starvation (2/2)

- So, what's a deadlock?
  - *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause*.
  - We will cover the deadlock in the next chapter.
- Another problem related to deadlocks is *starvation* or *indefinite blocking*.
  - A situation where processes wait indefinitely within the semaphore.
  - For example, the previous solution to the 'readers-writers' problem may result in starvation.
  - The starvation can be avoided by using a FCFS resource allocation policy.

Questions

- Any questions?

Outline

High-level language synchronization constructs

High-level language synchronization constructs

- As you can see, one subtle error when using semaphores may result in race conditions, deadlocks or other unpredictable and irreproducible behavior.

- As you can see, one subtle error when using semaphores may result in race conditions, deadlocks or other unpredictable and irreproducible behavior.
- To make the life of programmers easier, several high-level language synchronization constructs have been introduced.

High-level language synchronization constructs

- As you can see, one subtle error when using semaphores may result in race conditions, deadlocks or other unpredictable and irreproducible behavior.
- To make the life of programmers easier, several high-level language synchronization constructs have been introduced.
    - *Monitors*.

High-level language synchronization constructs

- As you can see, one subtle error when using semaphores may result in race conditions, deadlocks or other unpredictable and irreproducible behavior.
- To make the life of programmers easier, several high-level language synchronization constructs have been introduced.
    - *Monitors*.
        - It's suggested by Brinch-Hansen in 1973.

High-level language synchronization constructs

- As you can see, one subtle error when using semaphores may result in race conditions, deadlocks or other unpredictable and irreproducible behavior.
- To make the life of programmers easier, several high-level language synchronization constructs have been introduced.
    - *Monitors*.
        - It's suggested by Brinch-Hansen in 1973.
    - And many others...

Monitor

Monitor

- What's a monitor?

Monitor

- What's a monitor?
  - A monitor is a collection of procedures, variables and data structures that are all grouped together in a special kind of module or package.

Monitor

- What's a monitor?
  - A monitor is a collection of procedures, variables and data structures that are all grouped together in a special kind of module or package.
  - Monitors have an important property that makes them useful for achieving mutual exclusion: *only one process can be active in a monitor at any instant*.

Monitor

- What's a monitor?
  - A monitor is a collection of procedures, variables and data structures that are all grouped together in a special kind of module or package.
  - Monitors have an important property that makes them useful for achieving mutual exclusion: *only one process can be active in a monitor at any instant*.

```
monitor producer−consumer {
  int i;
  condition c;

  void produce()
  {
    ...
  }

  void consume()
  {
    ...
  }
}
```

Condition variables

- The monitor introduced so far is not enough for a reasonable solution.

Condition variables

- The monitor introduced so far is not enough for a reasonable solution.
    - We also need a way for processes to block when they cannot proceed within a monitor.

Condition variables

- The monitor introduced so far is not enough for a reasonable solution.
  - We also need a way for processes to block when they cannot proceed within a monitor.
- *Condition variables*

Condition variables

- The monitor introduced so far is not enough for a reasonable solution.
  - We also need a way for processes to block when they cannot proceed within a monitor.
- *Condition variables*
  - Condition variable is defined as a type of variables, which is associated with **only** two operations:

- The monitor introduced so far is not enough for a reasonable solution.
  - We also need a way for processes to block when they cannot proceed within a monitor.
- *Condition variables*
  - Condition variable is defined as a type of variables, which is associated with **only** two operations:
    - *c.wait();*

Condition variables

- The monitor introduced so far is not enough for a reasonable solution.
  - We also need a way for processes to block when they cannot proceed within a monitor.
- *Condition variables*
  - Condition variable is defined as a type of variables, which is associated with **only** two operations:
    - *c.wait();* - means that the process invoking it will be suspended until another process invokes

Condition variables

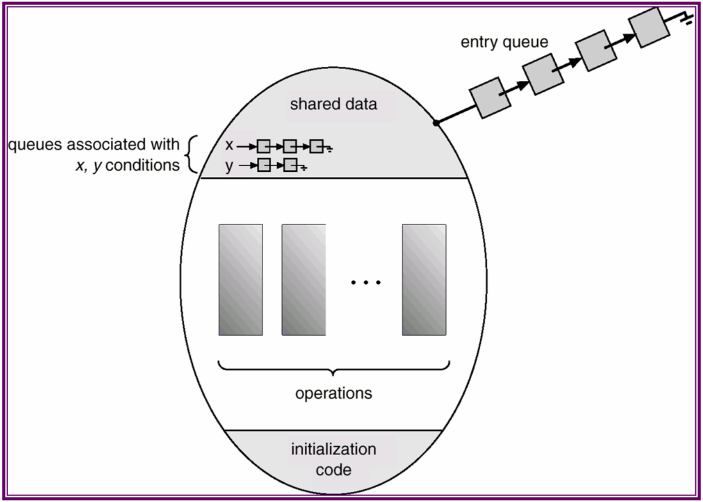
- The monitor introduced so far is not enough for a reasonable solution.
  - We also need a way for processes to block when they cannot proceed within a monitor.
- *Condition variables*
  - Condition variable is defined as a type of variables, which is associated with **only** two operations:
    - *c.wait();* - means that the process invoking it will be suspended until another process invokes
    - *c.signal();*

Condition variables

- The monitor introduced so far is not enough for a reasonable solution.
    - We also need a way for processes to block when they cannot proceed within a monitor.
- *Condition variables*
    - Condition variable is defined as a type of variables, which is associated with **only** two operations:
        - *c.wait();* - means that the process invoking it will be suspended until another process invokes
        - *c.signal();*
    - Note that if no process is suspended, then the *signal* operation has no effect.

Monitor with condition variables

Monitor with condition variables



What happens after a *c.signal()*?

What happens after a *c.signal()*?

- Suppose a process P is invoking *c.signal()* and another process Q is blocked by the condition variable c.

What happens after a *c.signal()*?

- Suppose a process P is invoking *c.signal()* and another process Q is blocked by the condition variable c.
- After P completed the *c.signal()*, it's possible that both P and Q will be active simultaneously within the monitor.

What happens after a *c.signal()*?

- Suppose a process P is invoking *c.signal()* and another process Q is blocked by the condition variable c.
- After P completed the *c.signal()*, it's possible that both P and Q will be active simultaneously within the monitor.
  - This will break the property of the monitor!

What happens after a *c.signal()*?

- Suppose a process P is invoking *c.signal()* and another process Q is blocked by the condition variable c.
- After P completed the *c.signal()*, it's possible that both P and Q will be active simultaneously within the monitor.
  - This will break the property of the monitor!
- Three possibilities exist:

What happens after a *c.signal()*?

- Suppose a process P is invoking *c.signal()* and another process Q is blocked by the condition variable c.
- After P completed the *c.signal()*, it's possible that both P and Q will be active simultaneously within the monitor.
  - This will break the property of the monitor!
- Three possibilities exist:
  - Hoare-style: suspending P and letting Q run.

What happens after a *c.signal()*?

- Suppose a process P is invoking *c.signal()* and another process Q is blocked by the condition variable c.
- After P completed the *c.signal()*, it's possible that both P and Q will be active simultaneously within the monitor.
  - This will break the property of the monitor!
- Three possibilities exist:
  - Hoare-style: suspending P and letting Q run.
  - Brinch-Hansen-style: P must leave the monitor immediately.

What happens after a *c.signal()*?

- Suppose a process P is invoking *c.signal()* and another process Q is blocked by the condition variable c.
- After P completed the *c.signal()*, it's possible that both P and Q will be active simultaneously within the monitor.
  - This will break the property of the monitor!
- Three possibilities exist:
  - Hoare-style: suspending P and letting Q run.
  - Brinch-Hansen-style: P must leave the monitor immediately.
  - Mesa-style (Mesa is a programming language): letting P run and suspending Q.

Monitor solution to the dining-philosopher problem

Monitor solution to the dining-philosopher problem

```
monitor dp {
  enum {THINKING, HUNGRY, EATING} state[N];
  condition c[N];

  void take_forks(int i) {
    state[i] = HUNGRY;
    test(i);
    if(state[i] != EATING)
      c[i].wait();
  }

  void test(int i) {
    if(state[i] == HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING) {
      state[i] = EATING;
      c[i].signal();
    }
  }
  void init() {
    for(int i = 0; i < N; i++)
      state[i] = THINKING;
  }
}
```

```
void put_forks(int i) {
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
}
```

Questions

- Any questions?

Language support

Language support

- The monitor constructs must be supported by the programming language to be useful.

Language support

- The monitor constructs must be supported by the programming language to be useful.
  - That is, the compiler must recognize the monitor construct and generate codes to support its functionality.

Language support

- The monitor constructs must be supported by the programming language to be useful.
  - That is, the compiler must recognize the monitor construct and generate codes to support its functionality.
- Example: Java (Mesa-style monitor with only one condition variable)

Language support

- The monitor constructs must be supported by the programming language to be useful.
  - That is, the compiler must recognize the monitor construct and generate codes to support its functionality.
- Example: Java (Mesa-style monitor with only one condition variable)
  - By adding the keyword *synchronized* to a method declaration, Java guarantees that once any thread has started executing that method, no other thread will be allowed to start executing any other *synchronized* method in that class.

Language support

- The monitor constructs must be supported by the programming language to be useful.
  - That is, the compiler must recognize the monitor construct and generate codes to support its functionality.
- Example: Java (Mesa-style monitor with only one condition variable)
  - By adding the keyword *synchronized* to a method declaration, Java guarantees that once any thread has started executing that method, no other thread will be allowed to start executing any other *synchronized* method in that class.
  - And Java provides two operations: *wait* and *notify* to block and wakeup the thread.

Questions

- Any questions?

Outline

Relationship of semaphore and monitor

Relationship of semaphore and monitor

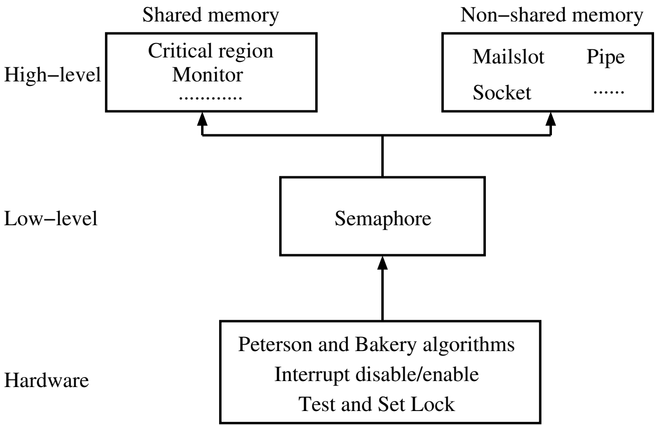
- Semaphore and monitor are *equivalent* in their functionality.

Relationship of semaphore and monitor

- Semaphore and monitor are *equivalent* in their functionality.
  - But their use and implementation are quite different.

Relationship of semaphore and monitor

- Semaphore and monitor are *equivalent* in their functionality.
  - But their use and implementation are quite different.



Implement semaphore using monitor

Implement semaphore using monitor

```
monitor semaphore {
  int value;
  condition c;

  void P() {
    value--;
    if(value < 0)
      c.wait();
  }

  void V() {
    value++;
    if(value <= 0)
      c.signal();
  }
}
```

Implement monitor using semaphore

Implement monitor using semaphore

```
semaphore mutex = 1, next = 0;
int next_count = 0;

// for each condition variable x
semaphore x_sem = 1;
int x_count = 0;
```

Implement monitor using semaphore

```
semaphore mutex = 1, next = 0;
int next_count = 0;

// for each condition variable x
semaphore x_sem = 1;
int x_count = 0;
```

```
// Mutual exclusion
// within a monitor
P(&mutex);
...
body of F;
...
if(next_count > 0)
  V(&next);
else
  V(&mutex);
```

Implement monitor using semaphore

```
semaphore mutex = 1, next = 0;
int next_count = 0;

// for each condition variable x
semaphore x_sem = 1;
int x_count = 0;
```

```
// Mutual exclusion
// within a monitor
P(&mutex);
...
body of F;
...
if(next_count > 0)
  V(&next);
else
  V(&mutex);
```

```
// x.wait()
x_count++;
if(next_count > 0)
  V(&next);
else
  V(&mutex);
P(&x_sem);
x_count--;
```

Implement monitor using semaphore

```
semaphore mutex = 1, next = 0;
int next_count = 0;

// for each condition variable x
semaphore x_sem = 1;
int x_count = 0;
```

```
// Mutual exclusion
// within a monitor
P(&mutex);
...
body of F;
...
if(next_count > 0)
  V(&next);
else
  V(&mutex);
```

```
// x.wait()
x_count++;
if(next_count > 0)
  V(&next);
else
  V(&mutex);
P(&x_sem);
x_count--;
```

```
// x.signal()
if(x_count > 0) {
   next_count++;
  V(&x_sem);
  P(&next);
   next_count--;
}
```

Questions

- Any questions?