Process Synchronization (Part II)

Amir H. Payberah
amir@sics.se

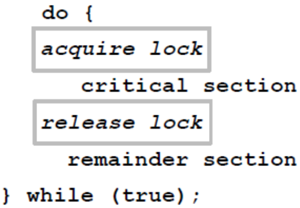
Amirkabir University of Technology
(Tehran Polytechnic)

Motivation

Process Synchronization

- Maintaining consistency of shared data
- Critical-Section (CS) problem
- CS solutions:
  - Peterson's solution
  - Synchronization Hardware
  - Mutex lock

Mutex Lock



```
   do {
   acquire lock
      critical section
   release lock
     remainder section
} while (true);
```

```
acquire() {
  while (!available); /* busy wait */

  available = false;
}

release() {
  available = true;
}
```

Semaphores

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore `S`: integer variable.
- Accessed via two atomic operations: `wait()` and `signal()`

wait() and signal()

```
wait(S) {
  while (S <= 0); // busy wait

  S--;
}
```

```
signal(S) {
  S++;
}
```

Counting and Binary Semaphore

- **Counting semaphore**: integer value can range over an unrestricted domain.

Counting and Binary Semaphore

- Counting semaphore: integer value can range over an unrestricted domain.

- Binary semaphore: integer value can range only between 0 and 1.
  - Same as a mutex lock.

Semaphore Usage (1/2)

- Access control to a given resource consisting of a finite number of instances.

Semaphore Usage (1/2)

- Access control to a given resource consisting of a finite number of instances.
  - Initialize the semaphore to the number of available resources.

Semaphore Usage (1/2)

- Access control to a given resource consisting of a finite number of instances.
  - Initialize the semaphore to the number of available resources.
  - Call `wait()` before using a resource.

Semaphore Usage (1/2)

- Access control to a given resource consisting of a finite number of instances.
  - Initialize the semaphore to the number of available resources.
  - Call `wait()` before using a resource.
  - Call `signal()` after releasing a resource.

Semaphore Usage (1/2)

- Access control to a given resource consisting of a finite number of instances.
    - Initialize the semaphore to the number of available resources.
    - Call `wait()` before using a resource.
    - Call `signal()` after releasing a resource.
    - If `S = 0`: all resources are used, and processes that wish to use a resource will block until the count becomes greater than 0.

Semaphore Usage (2/2)

- Consider P_1 and P_2 that require `C1` to happen before `C2`.

Semaphore Usage (2/2)

- Consider P_1 and P_2 that require `C1` to happen before `C2`.
- Create a semaphore `S` initialized to 0.

```
// Process P1
C1;
signal(S);

// Process P2
wait(S);
C2;
```

Semaphore Usage (2/2)

- Consider P_1 and P_2 that require `C1` to happen before `C2`.
- Create a semaphore `S` initialized to 0.

```
// Process P1
C1;
signal(S);

// Process P2
wait(S);
C2;
```

- The implementation still suffers from busy waiting.

Semaphore Implementation with no Busy Waiting (1/2)

- With each semaphore there is an associated waiting queue.

Semaphore Implementation with no Busy Waiting (1/2)

- With each semaphore there is an associated waiting queue.
- Each entry in a waiting queue has two data items:
    - Value (of type integer).
    - Pointer to next record in the list.

Semaphore Implementation with no Busy Waiting (1/2)

- With each semaphore there is an associated waiting queue.
- Each entry in a waiting queue has two data items:
  - Value (of type integer).
  - Pointer to next record in the list.

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

Semaphore Implementation with no Busy Waiting (2/2)

- `block`: place the process invoking the operation on the appropriate waiting queue.
- `wakeup`: remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy Waiting (2/2)

- `block`: place the process invoking the operation on the appropriate waiting queue.
- `wakeup`: remove one of processes in the waiting queue and place it in the ready queue.

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    // add this process to S->list;
    block();
  }
}
```

Semaphore Implementation with no Busy Waiting (2/2)

- `block`: place the process invoking the operation on the appropriate waiting queue.
- `wakeup`: remove one of processes in the waiting queue and place it in the ready queue.

```
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);
  }
}
```

Deadlock

- Deadlock: two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

Deadlock

- 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

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

P_1

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

Starvation

- Starvation: indefinite blocking.

Starvation

- Starvation: indefinite blocking.
- A process may never be removed from the semaphore queue in which it is suspended.

Starvation

- Starvation: indefinite blocking.

- A process may never be removed from the semaphore queue in which it is suspended.

- If we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Priority Inversion (1/2)

- Priority inversion: scheduling problem when lower-priority process holds a lock needed by higher-priority process.

Priority Inversion (1/2)

- Priority inversion: scheduling problem when lower-priority process holds a lock needed by higher-priority process.

- Example:
  - $L < M < H$, assume process H requires R, which is accessed by process L.

Priority Inversion (1/2)

- Priority inversion: scheduling problem when lower-priority process holds a lock needed by higher-priority process.
- Example:
    - $L < M < H$, assume process H requires R, which is accessed by process L.
    - Now suppose that process M becomes runnable, thereby preempting process L.

Priority Inversion (1/2)

- Priority inversion: scheduling problem when lower-priority process holds a lock needed by higher-priority process.

- Example:
  - $L < M < H$, assume process H requires R, which is accessed by process L.
  - Now suppose that process M becomes runnable, thereby preempting process L.
  - So, M has affected how long process H must wait.

Priority Inversion (2/2)

- Solved via priority-inheritance protocol.

Priority Inversion (2/2)

- Solved via priority-inheritance protocol.
    - All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.

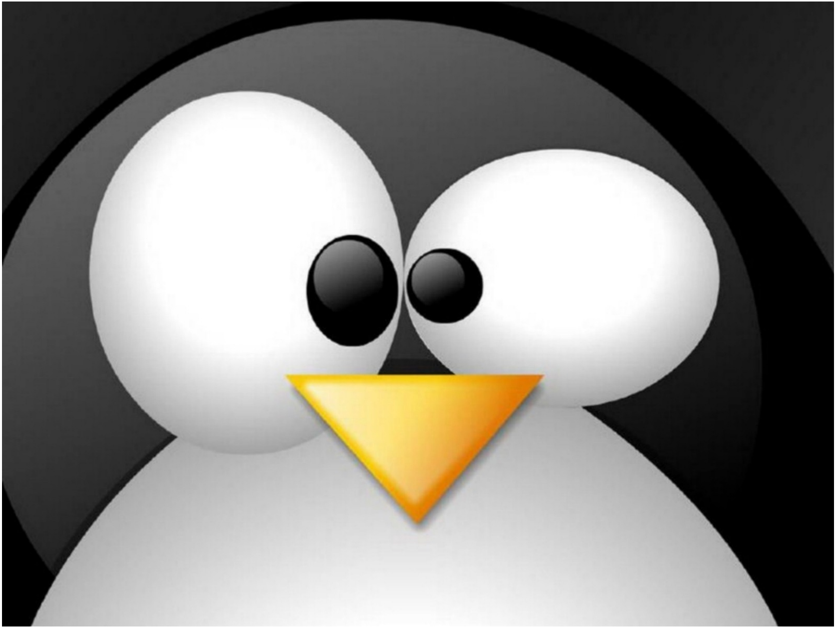
Priority Inversion (2/2)

- Solved via priority-inheritance protocol.
  - All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.
  - When they are finished, their priorities revert to their original values.

Priority Inversion (2/2)

- Solved via priority-inheritance protocol.
  - All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.
  - When they are finished, their priorities revert to their original values.

- Process L temporarily inherits the priority of process H, thereby preventing process M from preempting its execution.



Opening an Unnamed Semaphore

- `sem_init()` initializes a semaphore to the value specified by `value`.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Opening an Unnamed Semaphore

- `sem_init()` initializes a semaphore to the value specified by `value`.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- `pshared`: whether the semaphore is shared between threads or processes.
  - $== 0$: shared between the threads, and `sem` is the address of either a variable.
  - > 0: shared between processes, and `sem` is the address of a shared memory.

Destroying an Unnamed Semaphore

- `sem_destroy()` destroys the semaphore.

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

Waiting on a Semaphore

- `sem_wait()` decrements the value of the semaphore.

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

- value > 0: returns immediately.
- value $== 0$: blocks until the semaphore value rises above 0, then it decrements and `sem_wait()` returns.

Posting a Semaphore

- `sem_post()` increments the value of the semaphore.

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

- If the semaphore value was 0 before the `sem_post()`, and some other process is blocked waiting to decrement the semaphore, then that process is awoken.

Retrieving the Current Value of a Semaphore

- `sem_getvalue()` returns the current value of the semaphore.

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval);
```

Producer-Consumer Example

- Init the buffer and semaphore.

```
typedef struct {
    char buf[BSIZE];
    int nextin;
    int nextout;

    sem_t occupied;
    sem_t empty;
    sem_t mutex;
} buffer_t;

buffer_t buffer;

sem_init(&buffer.mutex, 0, 1);
sem_init(&buffer.occupied, 0, 0);
sem_init(&buffer.empty,0, BSIZE);
buffer.nextin = buffer.nextout = 0;
```

Producer-Consumer Example

- Producer

```
void producer(buffer_t *b, char item) {
    sem_wait(&b->empty);
    sem_wait(&b->mutex);

    b->buf[b->nextin] = item;
    b->nextin++;
    b->nextin %= BSIZE;

    sem_post(&b->mutex);
    sem_post(&b->occupied);
}
```

Producer-Consumer Example

- Consumer

```
char consumer(buffer_t *b) {
    char item;

    sem_wait(&b->occupied);
    sem_wait(&b->mutex);

    item = b->buf[b->nextout];
    b->nextout++;
    b->nextout %= BSIZE;

    sem_post(&b->mutex);
    sem_post(&b->empty);

    return(item);
}
```

Monitors

Problems with Semaphores

- Incorrect use of semaphore operations:

Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal(mutex)` ... `wait(mutex)`

Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal(mutex)` ... `wait(mutex)`
  - `wait(mutex)` ... `wait(mutex)`

Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal(mutex)` ... `wait(mutex)`
  - `wait(mutex)` ... `wait(mutex)`
  - Omitting of `wait(mutex)` or `signal(mutex)` (or both)

Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal(mutex)` ... `wait(mutex)`
  - `wait(mutex)` ... `wait(mutex)`
  - Omitting of `wait(mutex)` or `signal(mutex)` (or both)

- Deadlock and starvation are possible.

Monitors

- A high-level abstraction for process synchronization.

```
monitor monitor_name {

  /* shared variable declarations */
  function P1(... ) { ... }

  function P2(...) { ... }

  function Pn(...) { ... }

  initialization code(...) { ... }
}
```

Monitors

- A high-level abstraction for process synchronization.
- Abstract data type, internal variables only accessible by code within the procedure.

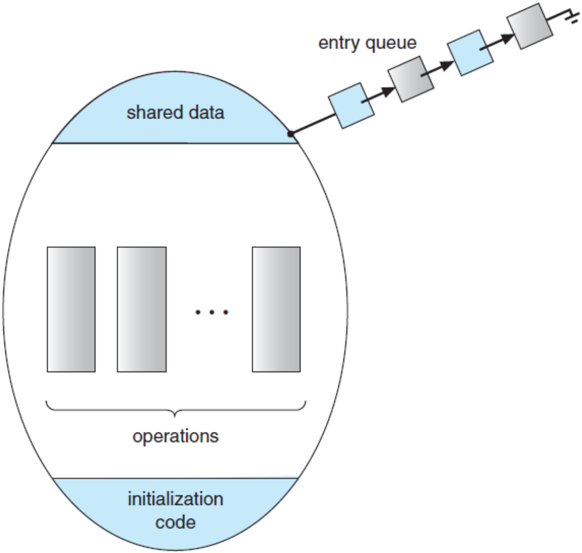
```
monitor monitor_name {

  /* shared variable declarations */
  function P1(... ) { ... }

  function P2(...) { ... }

  function Pn(...) { ... }

  initialization code(...) { ... }
}
```

Monitors

- A high-level abstraction for process synchronization.
- Abstract data type, internal variables only accessible by code within the procedure.
- Only one process may be active within the monitor at a time.

```
monitor monitor_name {

  /* shared variable declarations */
  function P1(... ) { ... }

  function P2(...) { ... }

  function Pn(...) { ... }

  initialization code(...) { ... }
}
```

A Monitor



Condition Variables

- `condition x, y;`

Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:

Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()`: a process that invokes the operation is suspended until `x.signal()`.

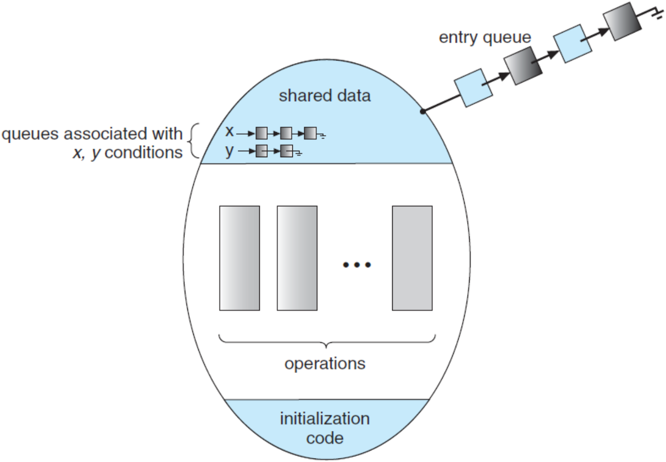
Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()`: a process that invokes the operation is suspended until `x.signal()`.
  - `x.signal()`: resumes one of processes (if any) that invoked `x.wait()`.

Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()`: a process that invokes the operation is suspended until `x.signal()`.
  - `x.signal()`: resumes one of processes (if any) that invoked `x.wait()`.
  - If no `x.wait()` on the variable, then it has no effect on the variable.

A Monitor with Condition Variables



Condition Variables Choices

- If process `P` invokes `x.signal()`, and process `Q` is suspended in `x.wait()`, what should happen next?

Condition Variables Choices

- If process `P` invokes `x.signal()`, and process `Q` is suspended in `x.wait()`, what should happen next?
    - Both `Q` and `P` cannot execute in parallel. If `Q` is resumed, then `P` must wait.

Condition Variables Choices

- If process `P` invokes `x.signal()`, and process `Q` is suspended in `x.wait()`, what should happen next?
  - Both `Q` and `P` cannot execute in parallel. If `Q` is resumed, then `P` must wait.

- Options include:

Condition Variables Choices

- If process `P` invokes `x.signal()`, and process `Q` is suspended in `x.wait()`, what should happen next?
    - Both `Q` and `P` cannot execute in parallel. If `Q` is resumed, then `P` must wait.

- Options include:
    - Signal and wait: `P` waits until `Q` either leaves the monitor or it waits for another condition.

Condition Variables Choices

- If process `P` invokes `x.signal()`, and process `Q` is suspended in `x.wait()`, what should happen next?
  - Both `Q` and `P` cannot execute in parallel. If `Q` is resumed, then `P` must wait.

- Options include:
  - Signal and wait: `P` waits until `Q` either leaves the monitor or it waits for another condition.
  - Signal and continue: `Q` waits until `P` either leaves the monitor or it waits for another condition.

Resuming Processes within a Monitor

- If several processes queued on condition `x`, and `x.signal()` executed, which should be resumed?

Resuming Processes within a Monitor

- If several processes queued on condition `x`, and `x.signal()` executed, which should be resumed?

- FCFS (First-Come, First-Served) frequently not adequate.

Resuming Processes within a Monitor

- If several processes queued on condition `x`, and `x.signal()` executed, which should be resumed?

- FCFS (First-Come, First-Served) frequently not adequate.

- Conditional-wait construct of the form `x.wait(c)`:
  - Where `c` is priority number.
  - Process with lowest number (highest priority) is scheduled next.

Single Resource Allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource.

```
R.acquire(t);
  ...
  access the resource;
  ...
R.release();
```

- Where `R` is an instance of type `ResourceAllocator` monitor.

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator {
  boolean busy;
  condition x;

  void acquire(int time) {
    if (busy)
      x.wait(time);
    busy = true;
  }

  void release() {
    busy = false;
    x.signal();
  }

  initialization code() {
    busy = false;
  }
}
```

Classical Problems of Synchronization

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

Bounded-Buffer Problem (1/3)

- `n` buffers, each can hold one item.
- Semaphore `mutex` initialized to the value `1`.
- Semaphore `full` initialized to the value `0`.
- Semaphore `empty` initialized to the value `n`.

Bounded-Buffer Problem (2/3)

- The structure of the producer process

```
do {
  ...
  /* produce an item in next produced */
  ...
  wait(empty);
  wait(mutex);
  ...
  /* add next produced to the buffer */
  ...
  signal(mutex);
  signal(full);
} while (true);
```

Bounded-Buffer Problem (3/3)

- The structure of the consumer process

```
do {
  wait(full);
  wait(mutex);
  ...
  /* remove an item from buffer to next consumed */
  ...
  signal(mutex);
  signal(empty);
  ...
  /* consume the item in next consumed */
  ...
} while (true);
```

Readers and Writers Problem

Readers and Writers Problem (1/3)

- A shared data set among a number of concurrent processes:
  - Readers: only read the data set; they do not perform any updates.
  - Writers: can both read and write.

Readers and Writers Problem (1/3)

- A shared data set among a number of concurrent processes:
  - Readers: only read the data set; they do not perform any updates.
  - Writers: can both read and write.
- Problem: allow multiple readers to read at the same time, only one single writer can access the shared data at the same time.

Readers and Writers Problem (1/3)

- A shared data set among a number of concurrent processes:
  - Readers: only read the data set; they do not perform any updates.
  - Writers: can both read and write.

- Problem: allow multiple readers to read at the same time, only one single writer can access the shared data at the same time.

- Shared Data
  - Semaphore `rw_mutex` initialized to 1.
  - Semaphore `mutex` initialized to 1.
  - Integer `read_count` initialized to 0.

Readers and Writers Problem (2/3)

- The writer process.

```
do {
  wait(rw_mutex);
  ...
  /* writing is performed */
  ...
  signal(rw_mutex);
} while (true);
```

Readers and Writers Problem (3/3)

- The reader process.

```
do {
  wait(mutex);
  read_count++;
  if (read_count == 1)
    wait(rw_mutex);
  signal(mutex);
  ...
  /* reading is performed */
  ...
  wait(mutex);
  read_count--;
  if (read_count == 0)
    signal(rw_mutex);
  signal(mutex);
} while (true);
```

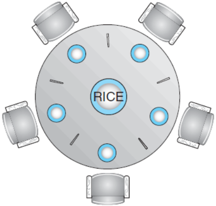
Dining-Philosophers Problem

Dining-Philosophers Problem (1/3)

- Philosophers spend their lives alternating thinking and eating.
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl.
- Need both to eat, then release both when done.

Dining-Philosophers Problem (1/3)

- Philosophers spend their lives alternating thinking and eating.
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl.
- Need both to eat, then release both when done.
- In the case of 5 philosophers:
  - Shared data: bowl of rice (data set)
  - Shared data: semaphore `chopstick[5]` initialized to 1



Dining-Philosophers Problem (2/3)

- The structure of philosopher `i`:

```
semaphore chopstick[5];

do {
  wait(chopstick[i]);
  wait(chopstick[(i+1) % 5]);
  ...
  /* eat for awhile */
  ...
  signal(chopstick[i]);
  signal(chopstick[(i+1) % 5]);
  ...
  /* think for awhile */
  ...
} while (true);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem (3/3)

- Deadlock handling

Dining-Philosophers Problem (3/3)

- Deadlock handling
  - At most 4 philosophers to sit simultaneously.

Dining-Philosophers Problem (3/3)

- Deadlock handling
  - At most 4 philosophers to sit simultaneously.
  - Allow a philosopher to pick up the forks only if both are available.

Dining-Philosophers Problem (3/3)

- Deadlock handling
  - At most 4 philosophers to sit simultaneously.
  - Allow a philosopher to pick up the forks only if both are available.
  - Use an asymmetric solution: an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Dining-Philosophers with Monitor (1/3)

```
monitor DiningPhilosophers {
  enum {THINKING; HUNGRY, EATING} state[5];

  condition self[5];

  initialization_code() {
    for (int i = 0; i < 5; i++)
    state[i] = THINKING;
  }

  void test(int i) {
    if ((state[i] == HUNGRY) &&
      (state[(i + 4) % 5] != EATING) &&
      (state[(i + 1) % 5] != EATING)) {
      state[i] = EATING ;
      self[i].signal() ;
    }
  }
```

Dining-Philosophers with Monitor (2/3)

```
  void pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING) self[i].wait;
  }

  void putdown(int i) {
    state[i] = THINKING;
    // test left and right neighbors
    test((i + 4) % 5);
    test((i + 1) % 5);
  }
}
```

Dining-Philosophers with Monitor (3/3)

- Each philosopher `i` invokes the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);
  EAT
DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible.

Summary

Summary

- Semaphore: counting semaphore and binary semaphore

Summary

- Semaphore: counting semaphore and binary semaphore
- Waiting queue to prevent busy waiting

Summary

- Semaphore: counting semaphore and binary semaphore
- Waiting queue to prevent busy waiting
- Deadlock and starvation

Summary

- Semaphore: counting semaphore and binary semaphore
- Waiting queue to prevent busy waiting
- Deadlock and starvation
- Priority inversion

Summary

- Semaphore: counting semaphore and binary semaphore
- Waiting queue to prevent busy waiting
- Deadlock and starvation
- Priority inversion
- Monitor: a high-level abstraction

Summary

- Semaphore: counting semaphore and binary semaphore
- Waiting queue to prevent busy waiting
- Deadlock and starvation
- Priority inversion
- Monitor: a high-level abstraction
- Classical problems: bounded-buffer, reader/writer, dining philosopher

Questions?

Acknowledgements

Some slides were derived from Avi Silberschatz slides.