Synchronization Methods in Shared Memory Model

Mutual Exclusion and Synchronization

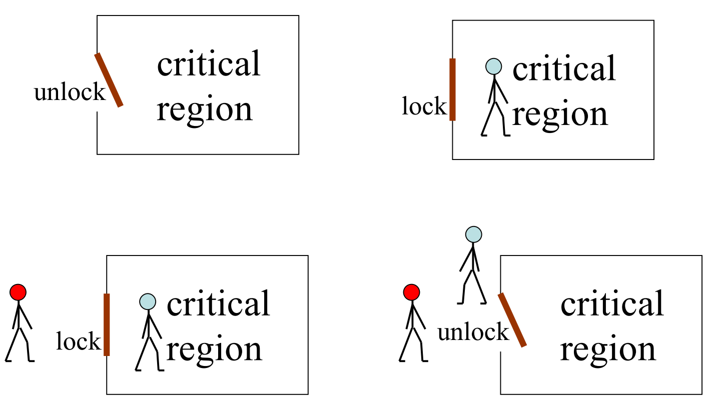
- Mutual exclusion
  - Guarantee that no two processes (or other types of agents) should access a critical region at the same time
- Synchronization
  - Guarantee that one step should occur before/after another
  - E.g., P_1 and P_2 run concurrently

    P_1: S_{11}, S_{12}

    P_2: S_{21}, S_{22}

    Guarantee that S_{12} executes before S_{22}
- Use **lock, semaphore, or monitor** to achieve mutual execution and/or synchronization

Critical Region and Locking



Simply set the lock -- Test-Set

- Consider two processes: A and B

  Process A and B: *lock* is initialized to *false*

  while *lock* **do** nothing;
  lock := *true*;
  < critical section >
  lock := *false*;

  - Execution

    A: **while** *lock* **do** nothing; -- find *lock* = *false*
    B: **while** *lock* **do** nothing; -- find *lock* = *false*
    A: *lock* := *true*; B: *lock* := *true*;
    A: < critical section > B: < critical section >

  - does not guarantee mutual exclusion at all

Too bad!!!

Simply set the lock -- Set-Test

- Consider two processes: A and B

  Process A and B:

  lock := *true*;
  while *lock* **do** nothing;
  < critical section >
  lock := *false*;

- Execution

  A: *lock* := *true*; B: *lock* := *true*;
  A: **while** *lock* **do** nothing; -- find *lock* = *true*
  B: **while** *lock* **do** nothing; -- find *lock* = *true*

- Deadlock, neither can enter CS

Still bad!!!

Test-Set, Use separate locks

Process A:	Process B:
while *lock*[*B*] **do** nothing;	**while** *lock*[*A*] **do** nothing;
lock[*A*] := *true*;	*lock*[*B*] := *true*;
< critical section >	< critical section >
lock[*A*] := *false*;	*lock*[*B*] := *false*;

Initialization: *lock*[*A*] = *lock*[*B*] = *false*

- Execution

  A: **while** *lock*[*B*] **do** nothing; -- find *lock*[*B*] = *false*
  B: **while** *lock*[*A*] **do** nothing; -- find *lock*[*A*] = *false*
  A: *lock*[*A*] := *true*;
  B: *lock*[*B*] := *true*;
  A: < critical section > B: < critical section >

- Similar to single lock algorithm, does not guarantee mutual exclusion

Set-Test, Use separate locks

Process A:
lock[*A*] := *true*;
while *lock*[*B*] **do** nothing;
< critical section >
lock[*A*] := *false*;

$lock[B] = F \Rightarrow$ no competition $\Rightarrow$ Enter CS

Process B:
lock[*B*] := *true*;
while *lock*[*A*] **do** nothing;
< critical section >
lock[*B*] := *false*;

- Execution
  - A: *lock*[*A*] := *true*;
  - B: *lock*[*B*] := *true*;
  - A: **while** *lock*[*B*] **do** nothing; -- find *lock*[*B*] = *true*
  - B: **while** *lock*[*A*] **do** nothing; -- find *lock*[*A*] = *true*
- Similar to the single lock case, deadlock, neither can enter
- But, if there is no completion, no problem

Improved from the single lock case

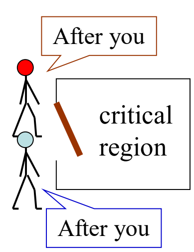
Set-Test, Separate lock, Give away

Process A:
lock[*A*] := *true*;
while *lock*[*B*] **do** ~~nothing~~
{ *lock*[*A*] := *false;*
delay (*short_time*);
lock[*A*] := *true;* }
< critical section >
lock[*A*] := *false*;

Give away on competition

Process B:
lock[*B*] := *true*;
while *lock*[*A*] **do** {
lock[*B*] := *false*;
delay (*short_time*);
lock[*B*] := *true;* }
< critical section >
lock[*B*] := *false*;

- Execution
  - A: *lock*[*A*] := *true*; B: *lock*[*B*] := *true*;
  - A: **while** *lock*[*B*] **do** -- find *lock*[*B*] = *true*
  - B: **while** *lock*[*A*] **do** -- find *lock*[*A*] = *true*
  - A: *lock*[*A*] := *false*; B: *lock*[*B*] := *false*;
- Close, but has mutual courtesy problem



Take turns

- Instead of using two locks, use one turn variable

Process A:
while *turn* != *A* **do** nothing;
< critical section >
turn := *B*;

Process B:
while *turn* != *B* **do** nothing;
< critical section >
turn := *A*;

- Guarantees mutual exclusion
- But processes have to take turns to enter CS
  - If one process does not have its turn (failed or not needing it), the others will not get it
  - More processes → taking turn is a bigger problem
- Close to a solution

Set-Test, Separate lock + Take turns

Original code for **Take turns**

Process A:
while *turn* != *A* **do** nothing;
< critical section >
turn := *B*;

Original code for
Set-Test, separate lock, give away

Process A:
lock[*A*] := *true*;

while *lock*[*B*] do

{ *lock*[*A*] := *false*;
delay (short time);
lock[*A*] := *true*;
}
< critical section >

lock[*A*] := *false*;

if *lock*[*B*] = *false*
⇒ no competition
⇒ Enter CS

When compete
⇒ Take turns

Set-Test, Separate lock + Take turns

Original code for **Take turns**

Process A:
  while *turn* != *A* **do** nothing;
  < critical section >
  turn := *B*;

Original code for **Take turns**
Set-Test, separate lock, ~~give away~~

Process A:
  lock[*A*] := *true*;

  while ***lock*[*B*] do**
   check turn
   { *lock*[*A*] := *false*;
    delay (short time);
    lock[*A*] := *true*;
   }
  < critical section >
  ***turn* := *B*;**
  lock[*A*] := *false*;

if *lock*[*B*] = *false*
⇒ no competition
⇒ Enter CS

When compete
⇒ Take turns

if *turn* = *A* (lock[A] has been set)
 (B cannot proceed) ⇒ A goes into CS
if *turn* != *A*, but **!*lock*[*B*]**
 (B does not compete) ⇒ A goes into CS

else loop till *lock*[*B*] = *false* or *turn* = *A*
 (after B finishes, this will be true)

Set-Test, Separate lock + Take turns

Original code for **Take turns**

Process A:
  while *turn* != *A* **do** nothing;
  < critical section >
  turn := *B*;

Original code for **Take turns**
Set-Test, separate lock, ~~give away~~

Process A:
  lock[*A*] := *true*;

  while *lock*[*B*]
   and *turn* != *A* do
   { ~~*lock*[*A*] := *false*;~~
    ~~delay (short time);~~
    ~~*lock*[*A*] := *true*;~~
   }
  < critical section >
  ***turn* := *B*;**
  lock[*A*] := *false*;

No need to release *lock*[*A*]

When *turn* = *B*, *B* proceeds anyway

if *turn* = *A*
 (B cannot proceed) ⇒ A goes into CS
if *turn* != *A*, but !*lock*[*B*]
 (B does not compete) ⇒ A goes into CS

else loop till *lock*[*B*] = *false* or *turn* = *A*
 (after B finishes, this will be true)

Set-Test, Separate lock + Take turns

Original code for **Take turns**
Set-Test, separate lock, ~~give away~~

Process A:
lock[*A*] := *true*;
***turn* := *B*;**
while *lock*[*B*]
and *turn* != *A* do
{ *lock*[*A*] := *false*;
delay (short time);
lock[*A*] := *true*;
}
< critical section >
~~***turn* := *B*;**~~
lock[*A*] := *false*;

Move ***turn* := *B*** up
When B comes to compete CS
⇒ B sets ***turn* := *A***
⇒ Problem solved!!!

if *turn* = *A*
(B cannot proceed) ⇒ A goes into CS
if *turn* != *A*, but !*lock*[*B*]
(B does not compete) ⇒ A goes into CS
but B may set ***lock*[*B*]** and **turn was B**
else loop till *lock*[*B*] = *false* or *turn* = *A*
(after B finishes, this will be true)

Both enter CS

Set-Test, Separate lock + Take turns

Original code for **Take turns**
Set-Test, separate lock, ~~give away~~

Process A:
lock[*A*] := *true*;
***turn* := *B*;**
while *lock*[*B*]
and *turn* != *A* do
{ *lock*[*A*] := *false*;
delay (short time);
lock[*A*] := *true*;
}
< critical section >
~~***turn* := *B*;**~~
lock[*A*] := *false*;

Move ***turn* := *B*** up
When B comes to compete CS
⇒ B sets ***turn* := *A***
⇒ Problem solved!!!

Another benefit from the move
Both yield the turn
Later comer succeeds in yield
Earlier comer wins the turn

Simplification of Lock and Take Turns

Process A:
lock[*A*] := *true*;
turn := *B*;
while *lock*[*B*] **and**
(*turn* != *A*) **do** nothing;
< critical section >
lock[*A*] := *false*;

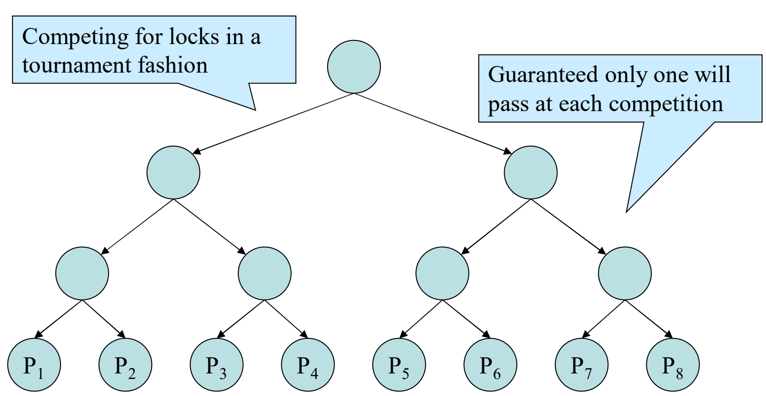
Process B:
lock[*B*] := *true*;
turn := *A*;
while *lock*[*A*] **and**
(*turn* != *B*) **do** nothing;
< critical section >
lock[*B*] := *false*;

This is the **Peterson's algorithm**

Before Peterson's algorithm, there was a similar but more complex solution, the **Dekker's algorithm**. But it is easier to understand this one by stepping to it directly.

- Software solutions
  - Quite significant overhead for a simple lock

Extend Lock for N Processes



Hardware Solutions

- Test and Set Instruction
  - In one instruction cycle: *Reg* → *lock* and *lock* → *Reg*
  - = swap (register, memory location)
- Use *Test-and-Set* for mutual exclusion

  R := *true*;

  repeat *Test-and-Set* (*R, lock*);

  until *R* = *false*;

  < critical section >;

  lock := *false*;

  tset instruction in x86

  - Similar to the first algorithm, but perform check and set in one instruction cycle

Use of Lock

- Examples
  - How to use *lock* for mutual exclusion?

    lock (*lck*);

    < critical section >

    unlock (*lck*);
  - How to use *lock* for synchronization?
    - E.g.: P_1: S_{11}, S_{12} P_2: S_{21}, S_{22}
      - Guarantee that S_{12} executes before S_{22}

        P_1: S_{11}; S_{12}; *unlock* (*lck*);

        P_2: S_{21}; *lock* (*lck*); S_{22}; *unlock* (*lck*);
  - What should be the initial value of *lck* (locked/unlocked)
- E.g.: A, B executed mutual exclusively, D after A, B

Use of Lock

- How to use *lock* for mutual exclusion?
  - Example: requirement
    - A and B executed mutual exclusively
    - D after A and B
  - Solution
    - lock(mutex); A; unlock(mutex); unlock(sync-a);
    - lock(mutex); B; unlock(mutex); unlock(sync-b);
    - lock(sync-a); lock(sync-b); D
      - How should the locks be initialized?
      - Can we use one sync lock?

Problem with Lock

- Software solutions are slow
- Hardware solutions
  - Much more efficient
  - Most of the systems support hardware solutions
    - Test-and-Set, Disable interrupts
- Lock problems in general
  - Require busy waiting
    - Also called spin lock
  - Unfair
    - First comer may not get the lock first
    - Potential starvation problem

Problem with Lock

- Busy waiting problem
  - Consider that P_1 is in critical section and P_2 waits for it
  - Single core systems
    - P_1 got switched out before finishing the critical section
    - P_2 got switched in, execute $lock$, start to wait on it
    - During P_2's time quantum, P_1 cannot execute and release $lock$ $\Rightarrow P_2$ will busy-wait on $lock$ till its time quantum expires
  - Multiple core systems
    - Same problem: P_1 can still get switched out no matter which processor it was running on and P_2 still will busy-wait on $lock$

P_1: …
<CS> switched out
…

P_2: …
lock… busy waiting
<CS>

Disable Interrupt

- How to solve the problem with lock
  - Disable Interrupts (another hardware solution)
    - Disable interrupts before CS and enable them after CS

      Disable interrupts;
      < critical section >
      Enable interrupts;
  - No process switch when executing CS
  - How about in multi-core systems?

IF in x86: interrupt flag
= 1 handle interrupts
= 0 disable interrupts

CLI instruction: clear IF
SLI instruction: set IF

But IF is only for hardware interrupts

CLI & SLI are privileged instructions (kernel only)

Semaphores

- How to solve the problems with locks
  - Use a queue to block processes waiting to enter CS
    - Avoid busy waiting: processes are blocked until its turn comes
    - Achieve fairness: queue enforces order
- Integer semaphore
  - Has a queue: to hold blocked processes
  - Has a counter
    - Control: > 0: pass; ≤ 0: blocked
    - Counting the number of processes in queue: ≤ 0
  - Offer Two functions: *wait* and *signal*
    - *wait*: request to enter the critical section - also *P*
    - *signal*: release the critical section - also *V*

Semaphore Implementation

- **type** *semaphore* =
    record *count*: *integer*;
            queue: **queue of** *processes*;
    end;
var *s*: *semaphore* := 10;

- *wait*(*s*):
    s.*count* := *s*.*count* – 1;
    if (*s*.*count* < 0) **then**
    { block process *P*;
      put *P* in *s*.*queue*;
    }

  signal(*s*):
    s.*count* := *s*.*count* + 1;
    if (*s*.*count* <= 0) **then**
    { remove *P* from *s*.*queue*;
      put *P* in ready list;
    }

Semaphore Implementation

A: load s.count -- s.count = 1
B: load s.count -- s.count = 1
A: add –1
B: add –1
A: (s.count < 0) → no → <pass semaphore wait>
B: (s.count < 0) → no → <pass semaphore wait>
Too bad!!!

- **type** *se*
  recor

  end;
  var *s*: *s*

- *wait*(*s*):
  s.count := *s.count* – 1;
  if (*s.count* < 0) **then**
  { block process *P*;
  put *P* in *s.queue*;
  }

  signal(*s*):
  s.count := *s.count* + 1;
  if (*s.count* <= 0) **then**
  { remove *P* from *s.queue*;
  put *P* in ready list;
  }

Semaphore Implementation

- Semaphore *wait* and *signal* functions are themselves critical sections
  - Simultaneous accesses to *s.count* and *s.queue* can cause problem
- Need to use lock to protect *wait* and *signal* functions
  - Lock solution has busy waiting problem
  - Busy waiting for small segments of code is not too much overhead (very low probability of getting switched out)

Semaphore Implementation

wait(*s*):

lock (*s.lck*);
s.count := *s.count* – 1;
if (*s.count* < 0) **then**
{ block process *P*;
put *P* in *s.queue*;
}
unlock (*s.lck*);

signal(*s*):

lock (*s.lck*);
s.count := *s.count* + 1;
if (*s.count* <= 0) **then**
{ remove *P* from *s.queue*;
put *P* in ready list;
}
unlock (*s.lck*);

Semaphore has
s.count, *s.queue*, and ***s.lck***

Use of Semaphores

- Use semaphores for mutual exclusion

  s.count := 1;
  wait (*s*);
  < critical section >
  signal (*s*);

- Use semaphores for synchronization
  - E.g., Guarantee that S_{12} executes before S_{22}

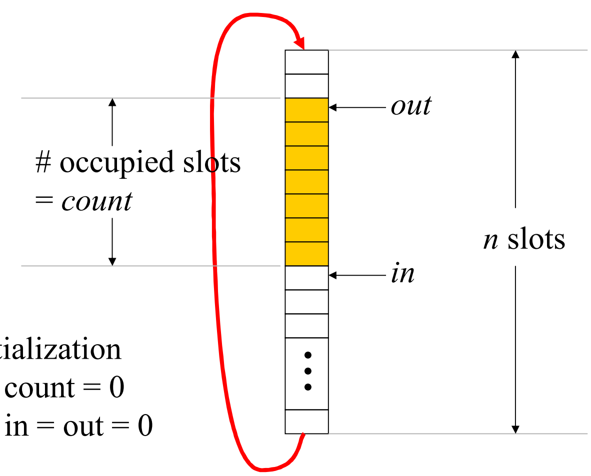
    P_1: S_{11}, S_{12} P_2: S_{21}, S_{22}

  s.count := 0;
  P_1: S_{11}, S_{12}, *signal* (*s*);
  P_2: S_{21}, *wait* (*s*); S_{22}

Use of Semaphores

- Typical semaphore programming examples
  - Bounded buffer problem
  - Reader-writer problem
  - Bakery problem

Bounded Buffer Problem



Initialization
int count = 0
int in = out = 0

Bounded Buffer Problem

process producer;
  repeat
    produce *item*;
    while *count* = *n* **do** nothing;
    buffer[*in*] := *item*;
    in := (*in*+1) mod *n*;
    count := *count* + 1;
  until forever

process consumer;
  repeat
    while *count* = 0 **do** nothing;
    item := *buffer*[*out*];
    out := (*out*+1) mod *n*;
    count := *count* – 1;
    consume *item*;
  until forever

Bounded Buffer Problem

process producer;
  repeat
    produce *item*;
    while *count* = *n* **do** nothing;
    buffer[*in*] := *item*;
    in := (*in*+1) mod *n*;
    count := *count* + 1;
  until forever

process consumer;
  repeat
    while *count* = 0 **do** nothing;
    item := *buffer*[*out*];
    out := (*out*+1) mod *n*;
    count := *count* – 1;
    consume *item*;
  until forever

If there are two producers
A1: buffer[5] := item_A;
A2: buffer[5] := item_B;
Put in the same slot!!!

Current: count = 9; n = 10
A: load count -- count = 9
B: load count -- count = 9
A: incr count   -- count = 10
B: decr count   -- count = 8
Wrong count!!!

Bounded Buffer Problem

process producer;
  repeat
    produce *item*;
    ***wait* (*mutex*);**
    while *count* = *n* **do** nothing;
    buffer[*in*] := *item*;
    in := (*in*+1) mod *n*;
    count := *count* + 1;
    ***signal* (*mutex*);**
  until forever

process consumer;
  repeat
    ***wait* (*mutex*);**
    while *count* = 0 **do** nothing;
    item := *buffer*[*out*];
    out := (*out*+1) mod *n*;
    count := *count* – 1;
    ***signal* (*mutex*);**
    consume *item*;
  until forever

Bounded Bu

Current: count = 10; n = 10;
P: get the lock
P: find count = 10, wait
⇒
Need to enter the critical section to change count
But no one can enter CS

process producer;
  repeat
    produce *item*;
    ***wait* (*mutex*);**
    while *count* = *n* **do** nothing;
    buffer[*in*] := *item*;
    in := (*in*+1) mod *n*;
    count := *count* + 1;
    ***signal* (*mutex*);**
  until forever

g;
    out := (*out*+1) mod *n*;
    count := *count* – 1;
    ***signal* (*mutex*);**
    consume *item*;
  until forever

Bounded Buffer Problem

```
process producer;
  repeat
    produce item;
    while count = n do nothing;
    wait (mutex);
    buffer[in] := item;
    in := (in+1) mod n;
    count := count + 1;
    signal (mutex);
  until forever
```

```
process consumer;
  repeat
    while count = 0 do nothing;
     wait (mutex);
    item := buffer[out];
    out := (out+1) mod n;
    count := count – 1;
     signal (mutex);
    consume item;
  until forever
```

Bounded Buffer Problem

```
process producer;
  repeat
    produce item;
    while count = n do nothing;
    wait (mutex);
    buffer[in] := item;
    in := (in+1) mod n;
    count := count + 1;
    signal (mutex);
  until forever
```

Current: count = 9; n = 10
P1: check (count = n)
-- count = 9, not full
P2: check (count = n)
-- count = 9, not full
P2: get the mutex
put item in buffer
count = count + 1;
-- buffer full
P1: checked earlier, buffer was not full, will not check again, but buffer is full, cause problem

⇒ Put the while loop in

until forever

Bounded Buffer Problem

- Check full or empty
  - Producer check "buffer full"
    - Ok to block other producers but should not block consumers
  - Consumer check "buffer empty"
    - Ok to block other consumers but should not block producers
- How about using different semaphores?
  - Update buffer/count: protected by the same semaphore
  - Check full: one semaphore to protect from all producers
  - Check empty: one semaphore to protect from all consumers

Bounded Buffer Problem

Use semaphores for conditional wait ⇒ Solve the problem + avoid busy waiting

process producer;
  repeat
    produce *item*;
    ~~**while** *count* = *n* **do** nothing;~~ wait on full condition
    ***wait* (*mutex*);**
    buffer[*in*] := *item*;
    in := (*in*+1) mod *n*;
    count := *count* + 1;
    ***signal* (*mutex*);** Signal empty (no longer empty)
  until forever

process consumer;
  repeat
    wait on empty condition
    ~~**while** *count* = 0 **do** nothing;~~
    ***wait* (*mutex*);**
    item := *buffer*[*out*];
    out := (*out*+1) mod *n*;
    count := *count* – 1;
    ***signal* (*mutex*);** Signal full (no longer full)
    consume *item*;
  until forever

Semaphores need to be signaled. When to signal???
When the condition no long holds!

Bounded Buffer Problem

Need to initialize 3 semaphores!

- Use semaphore count to do counting

```
process producer;
    loop
        produce item;
        wait (full);
        wait (mutex);
        access shared_buffer;
        signal (mutex);
        signal (empty);
    end
end;
```

```
process consumer;
    loop
        wait (empty);
        wait (mutex);
        access shared_buffer;
        signal (full);
        signal (mutex);
        consume item;
    end
end;
```

Bounded Buffer Problem

Need to initialize 3 semaphores!

- Use semaphore count to do counting

```
process producer;
    loop
        produce item;
        wait (full);
        wait (mutex);
        access shared_buffer;
        signal (mutex);
        signal (empty);
    end
end;
```

Init:
mutex = 1

```
process consumer;
    loop
        wait (empty);
        wait (mutex);
        access shared_buffer;
        signal (full);
        signal (mutex);
        consume item;
    end
end;
```

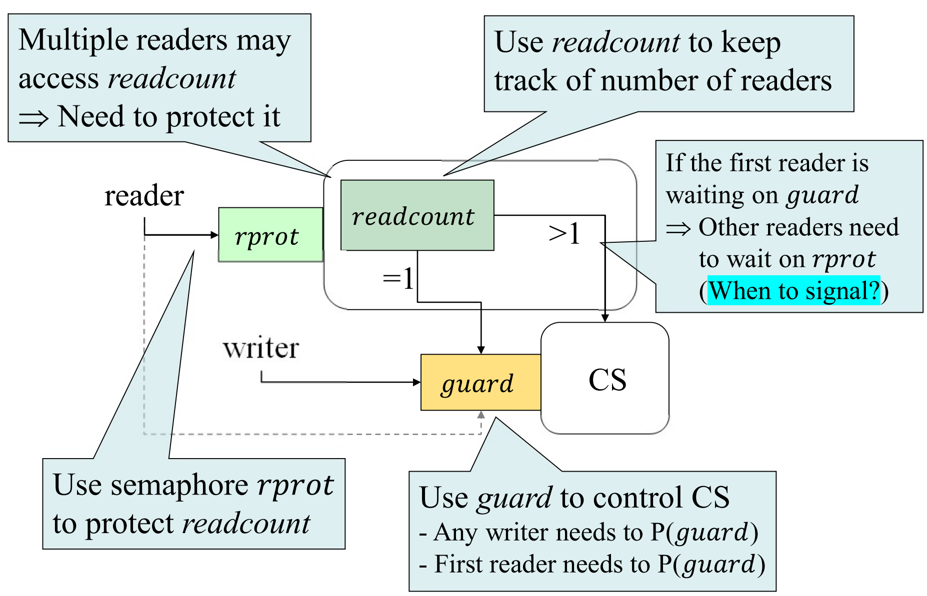
init: full = N
-- distance to full

init: empty = 0
-- distance to empty

Reader-Writer Problem

- Many readers can read at the same time without causing problem
- When there is a writer, it has to write exclusively
- First reader/writer problem (reader has priority)
  - If $\geq$ 1 reader in CS $\Rightarrow$ allow more readers to get in
  - If a writer is in CS, no one else can enter
  - This way readers has a higher priority
  - Have starvation problem for writers
- There are other forms of the reader/writer problem

First Reader-Writer Problem



First Reader-Writer Problem

- Entry time
  - $readcount = 1 \rightarrow$ first reader $\Rightarrow$ wait on $guard$
    - If there is a writer in CS: this reader waits on $guard$
    - After entering CS: signal $rprot$, to let other readers access readcount and CS
  - $readcount > 1 \rightarrow$ at least one reader is already in CS
    - Proceed to CS directly
- Exit time
  - $readcount > 0 \rightarrow$ more readers in CS $\Rightarrow$ Just leave
  - $readcount = 0 \rightarrow$ no more readers in CS $\Rightarrow$ signal $guard$
    - If there is a writer, signal $guard$ to let a writer proceed
    - Signal simply allows future entrance to CS

First Reader-Writer Problem

- Reader Process:

  wait (*rprot*);

  readcount := *readcount* + 1;

  if (*readcount* = 1) **then** *wait* (*guard*);

  signal (*rprot*); **Generally, never wait inside semaphore pairs**

  < enter critical section for read >

  wait (*rprot*);

  readcount := *readcount* – 1;

  if *readcount* = 0 **then** *signal* (*guard*);

  signal(*rprot*);

First Reader-Writer Problem

- Writer Process:

  wait (*guard*);

  < enter critical section for write >

  signal (*guard*);

Bakery Problem

- A bakery has N salesmen
- Customers can come at any time
  - No bound on number of customers
- A salesman waits for a customer and provide service
  - wait for customer; provide service;
- If no salesman is available, a customer has to wait
  - wait for salesman; get service;
- Write a program using semaphores to synchronize the salesmen and customers
  - cust: customer
  - sales: salesman

Bakery Problem

- Consider a different version first
  - There are N salesman and M customers

Salesman Process:
  repeat
    wait (*cust*);
    provide service;
    signal (*sales*);
  until *false*;

Customer Process:
  wait (*sales*);
  get service;
  signal (*cust*);

But
Customers are not the same and are of unknown number

Customer Process:
  signal (*cust*);
  wait (*sales*);
  get service;

Initialization
cust **= 0**
sales* = *N

Types of Semaphores

- Integer semaphore (general semaphore): has a count
  - What we used are integer semaphores
  - Most common type
- Binary semaphores
  - Does not keep count, semaphore value can only be 0 or 1
    - Similar to lock, but has a queue
  - In some cases, a binary semaphore is sufficient; e.g., mutex

Properties of Semaphores

- Lock
  - Has the busy waiting problem
- Semaphore
  - Requires context switch
- When the critical section is small
  - Short busy-wait may be cheaper than context switch $\Rightarrow$ use lock instead of semaphore
  - But remember the potential of: process in CS is switched out and other processes waiting on the lock will busy wait till time quantum expires
    - Hopefully the probability of this is very small
    - And, semaphore contains lock and has the same problem

Semaphores in Pthread

- Creating a semaphore

  #include <semaphore.h>

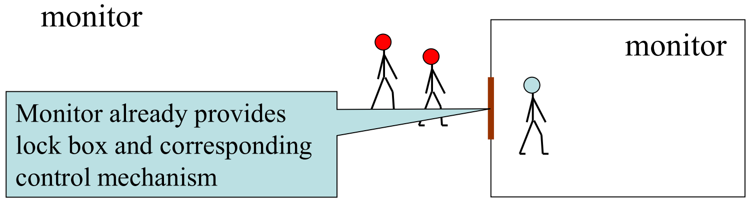
  sem_t sem;
- Operations on a semaphore
  - int sem_init (sem_t *sem, int pshared, unsigned int value);
    - pshared: allow the semaphore to be shared by multiple processes
      - General situations: set pshared to 0
    - value: initialization value for the semaphore
  - int sem_wait(sem_t * sem);
  - int sem_post(sem_t * sem);
  - Other: sem_getvalue, sem_destroy, sem_trywait
- Project 3

Properties of Semaphores

- Semaphore is a powerful primitive
  - Can do all types of synchronizations
  - however, similar to goto statement → unstructured
    - may issue wait and forget to signal → lock out other processes
    - may reverse the order of multiple waits → deadlock
- Hard to debug semaphore code
  - Wait and signal of semaphores may spread all over
  - Lack of encapsulation

Monitor

- Protect shared objects within an abstraction
  - Provide **encapsulation**
  - Accesses to a shared object is confined within a monitor
  - Easier to debug the code
- Provide mutual exclusive accesses
  - No two processes can be active at the same time within a monitor



Shared Device Program with Monitor

- N devices in the system
  - Use any of them as long as it is free

monitor mutex_devices:

free: **array** [0..N–1] **of** *boolean*; -- initialized to *true*

int *acquire* () {

for *i* := 0 **to** N–1 **do**

if (*free*[*i*]) **then** { *free*[*i*] := *false*; **return** (*i*); }

return (–1); }

Modify this to always return a free printer

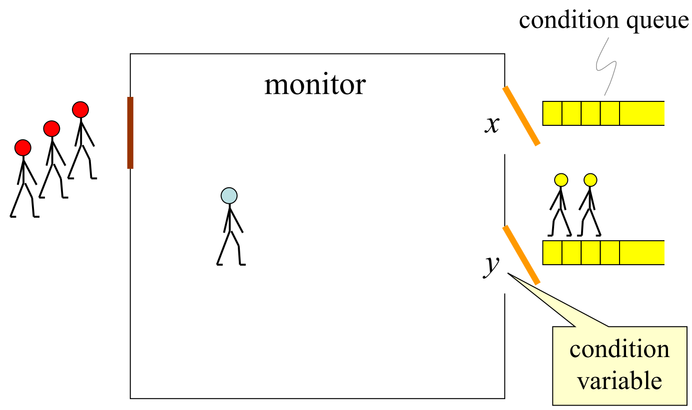
void *release* (*index*: *integer*)

{ *free*[*index*] := *true*; }

Synchronization within a Monitor

- Monitor guarantees mutual exclusive accesses
- May still need synchronization
  - E.g., bounded buffer problem
    - wait for buffer to become full/empty
- Monitor also provides condition variables
  - To achieve conditional wait
  - Associated with each condition variable
    - A condition queue
    - The *wait* and *signal* functions
  - Wait inside a monitor
    - Same as wait inside a lockbox protected by a semaphore (mutex)
    - Has potential of deadlock
    - Monitor provides mechanism to counter it

Synchronization within a Monitor



Bounded Buffer Problem with Monitor

monitor bounded_buffer
buffer: **array** [0..*n*-1] **of** *item*;
in, *out*, *counter*: *integer* := 0;
empty, *full*: *condition*;

In semaphore: P (full/empty)
In monitor: condition wait
??? Won't work

function *deposit* (*item*)
{ *full.wait*;
access buffer;
counter := *counter* + 1;
empty.signal;
}

function *remove* (&*item*)
{ *empty.wait*;
access buffer;
counter := *counter* – 1;
full.signal;
}

Bounded Buffer Problem with Monitor

In monitor:
Condition wait/signal are memoryless!!!

monitor bounded_buffer
  buffer: **array** [0..*n*-1] **of** *item*;
  in, *out*, *counter*: *integer* := 0;
  empty, *full*: *condition*;

function *deposit* (*item*)
{ **if** (*counter* = *n*) **then**
    full.wait;
  access buffer;
  counter := *counter* + 1;
  empty.signal;
}

function *remove* (&*item*)
{ **if** (*counter* = 0) **then**
    empty.wait;
  access buffer;
  counter := *counter* – 1;
  full.signal;
}

Bounded Buffer Problem with Monitor

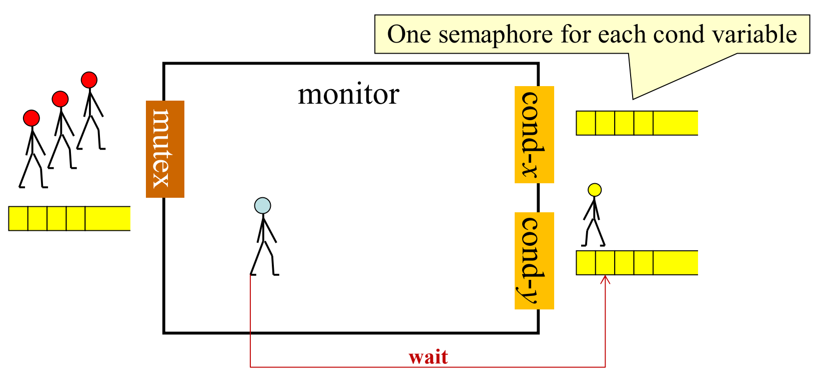
Producer process:
  repeat
    produce *item*
    deposit (*item*);
  until *false*;

Consumer process:
  repeat
    remove (*item*);
    consume *item*;
  until *false*;

Synchronization within a Monitor

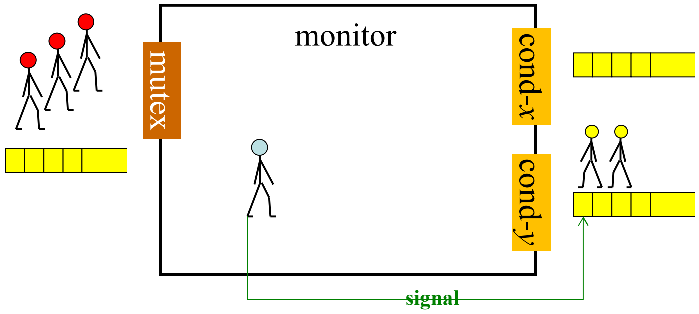
- Condition variables do not have counting ability
  - *cond.wait*:
    - The calling process always get blocked
    - Have to check the condition before doing *cond.wait*
  - *cond.signal*:
    - If no waiting process, the signal is discarded
- Semaphore has count (which controls sync)
  - Wait
    - Wait may not block the calling process, depending on count
    - Can do wait directly without checking
  - Signal is like being "recorded" since it increases count

Implementing Monitor with Semaphores



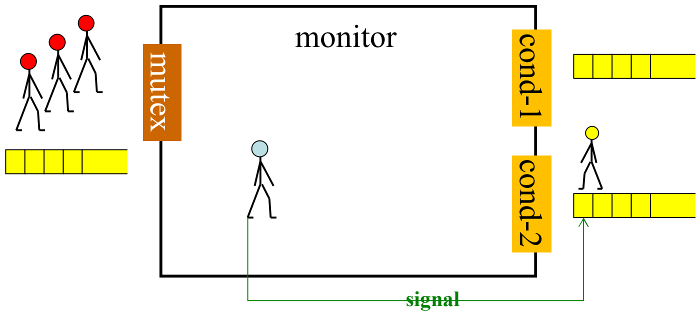
Semaphore "mutex" controls the monitor mutual exclusive entry
One binary semaphore for each condition variable
Implementation of cond.wait: signal(mutex); wait(cond);
-- This will be modified later
-- Also need to maintain counters to ensure correct implementation

Implementing Monitor with Semaphores



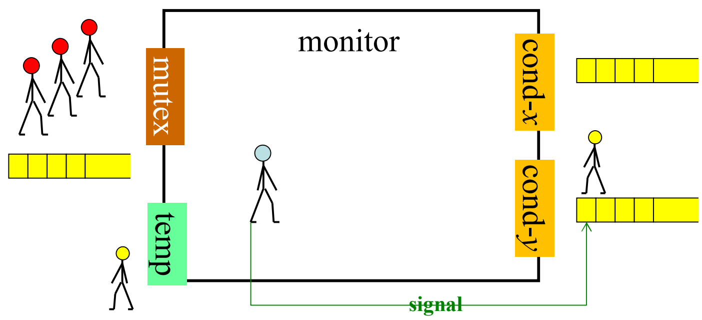
Monitor can only allow one active process within it.
Upon signal, who should be kicked out of the monitor,
signaler or signalee?

Implementing Monitor with Semaphores



Choose to let the signaler continue after cond1.signal
But condition cond1 may be changed by the signaler's subsequent code
⇒ when signalee comes in monitor, the wait condition should be checked again
(put extra burden on the programmer)
Benefit:
If signaler code does not change the condition ⇒ Save on context switch
If signaler is going to exit ⇒ Great, no need to incur the queuing overhead

Implementing Monitor with Semaphores



Choose to let the signalee continue after cond1.signal (common choice)
Signaler: moved to temp queue by temp.wait
Every process upon exiting the monitor ⇒ release signaler first
⇒ if temp.count > 0, temp.signal; else mutex.signal
*** temp keeps all waiting signalers (no need to be cond specific)

Monitor and Semaphore

- Monitor provides encapsulation
  - What should be encapsulated?
  - Too much ⇒ reduce concurrency
    - Some part of the code that can be executed concurrently, if encapsulated in the monitor, can cause reduced concurrency
    - If not encapsulate them, then lose the meaning of encapsulation
    - Reader/writer example
- If monitor is used to do what a semaphore would do
  - Monitor is more expensive

Monitor Deficiency

- Reduce concurrency

monitor rw_object
{ *shared_object*: some type;

function *read* ()
{ read and return some attributes of *shared_object*; }

function *write* (*x*)
{ write *x* to *shared_object*; }
}

Monitor Deficiency

```
Monitor rw_object
{ rcount: integer;  busy: boolean;
  ok2read, ok2write: condition;

  procedure startread;
    if busy then
      ok2read.wait;
    rcount := rcount + 1;
    ok2read.signal
  end;

  procedure startwrite;
    if busy or rcount != 0 then
      ok2write.wait;
    busy := true;
  end;

  procedure endread;
    rcount := rcount – 1;
    if rcount = 0 then
      ok2write.signal;
  end;

  procedure endwrite;
    busy := false;
    if empty (ok2read.queue) then
      ok2write.signal;
    else ok2read.signal;
  end;
```

CS is outside ⇒ Does not provide sufficient encapsulation!
⇒ Defeat the purpose of monitor

Readings

- All of Chapter 5