Process Synchronization (V)

Condition Variables

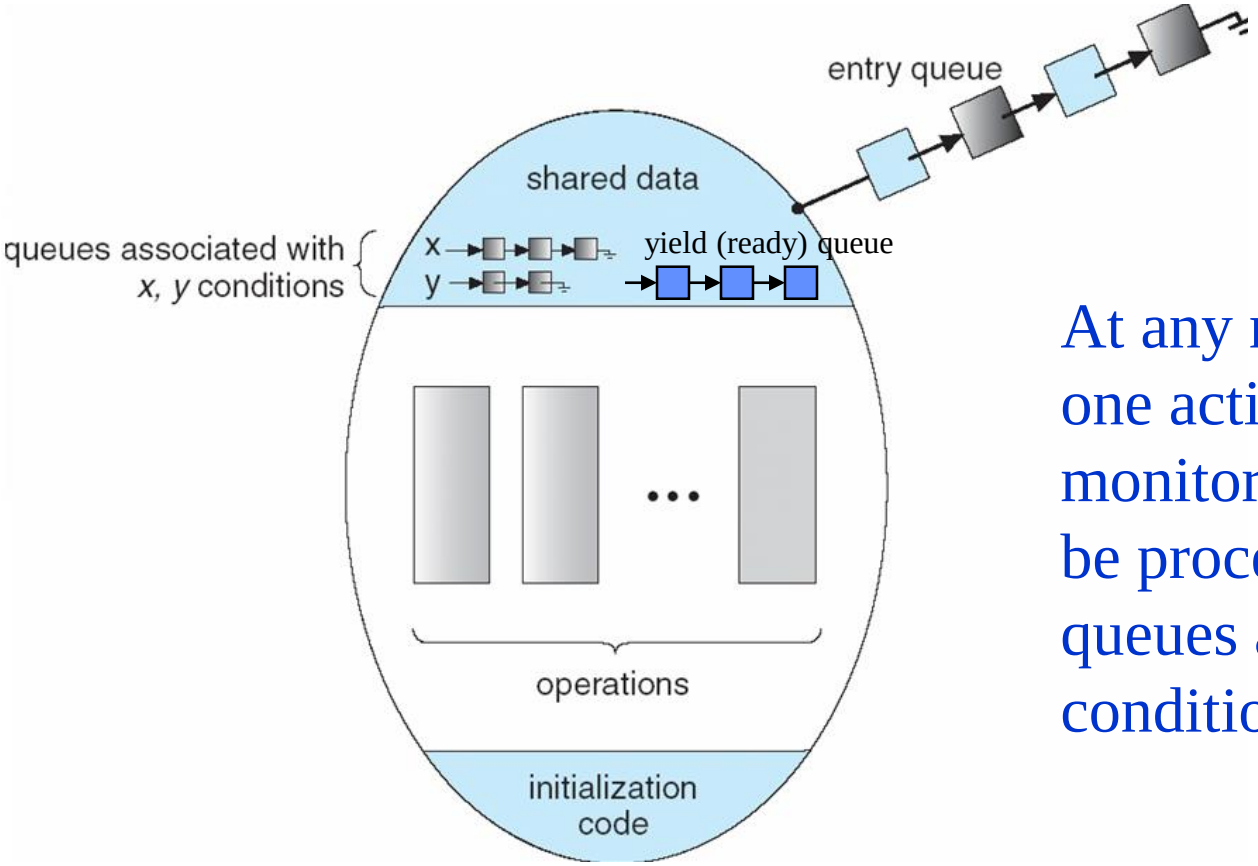
- Monitor can be used to realize mutual exclusion. How to realize more sophisticated synchronization?
- Condition variables
  - Condition variables can be defined in a monitor as defining regular variables

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

    x.wait( ) – a process that invokes the operation is suspended.

    x.signal( ) – resumes (wakes up) one of processes (if any) that invoked x.wait (), and blocks the calling process itself unless there is no active process in the monitor (i.e., yield to the woken-up process)

Monitor with Condition Variables



At any moment, at most one active process is in a monitor; but there could be processes blocked in queues associated with conditions or that yield.

Bounded-Buffer Problem

```
producers-consumers: monitor
begin
  in,out: integer
  pool: 0..9 of buffer;
  OKtoproduce, OKtoconsume:
    condition;
procedure produce (x: buffer);
begin
    if in >= out+10 then
     OKtoproduce.wait;
    pool[in mod 10] := x;
    in := in+1;
   OKtoconsume.signal;
end produce;
```

```
procedure consume(y: buffer);
begin
  if  out >= in then
    OKtoconsume.wait;
  y:= buffer[out mod 10];
  out := out+1;
  OKtoproduce.signal;
end consume;


begin (* initialization *)
  in, out :=0;
end;
end producers-consumers;
```

Reader-Writer's Problem

Is there any way to implement it within a single monitor? NO

```
Monitor Reader-Writer
Begin
        …

        Procedure Read
        begin
                …
        end

        Procedure Write
        begin
                …
        end
end
```

Reader-Writer's Problem

Writer Process

readers-writers.startwrite
Write to the file
readers-writers.endwrite

Reader Process

readers-writers.startread
Read the file
readers-writers.endread

Reader-Writer's Problem

```
Monitor Reader-Writer
begin
readercount: integer //# of reading readers
busy: boolean //if a writer is writing
OKtoread, OKtowrite: condition
procedure startread
begin
  If busy then OKtoread.wait;
  readercount := readercount + 1;
  OKtoread.signal
end startread
procedure endread;
begin
  readercount := readercount-1;
  If readercount = 0 then OKtowrite.signal;
end endread

procedure startwrite;
begin
   If busy or readercount <>0 then OKtowrite.wait;
   busy := true;
end startwrite

procedure endwrite;
begin
  busy := false
  if OKtoread.queue then OKtoread.signal
  else OKtowrite.signal
end endwrite

begin /* initialization of local data */
 readercount := 0;
 busy := false;
end
end readers-writers
```

```
procedure startread
begin
  If busy then OKtoread.wait;
  readercount := readercount + 1;
  OKtoread.signal
end startread

procedure endread;
begin
  readercount := readercount-1;
  If readercount = 0 then
OKtowrite.signal;
end endread

procedure startwrite;
begin
   If busy or readercount <>0 then OKtowrite.wait;
   busy := true;
end startwrite

procedure endwrite;
begin
  busy := false
  if OKtoread.queue then OKtoread.signal
  else OKtowrite.signal
end endwrite
```

busy: ~~F~~ **T**

readercount: ~~0~~ ~~1~~ ~~2~~ ~~1~~ **0**

OKtoread.queue	
OKtowrite.queue	~~W1~~
Yield queue	~~R2~~
Entry queue	~~W1~~ R1

Time line

R1 → If busy then OKtoread.wait
readercount := readercount + 1

W1 → OKtoread.signal

R1 reads shared file

If busy or readercount <>0 then
OKtowrite.wait

R2 → If busy then OKtoread.wait

readcount := readcount +1

R1 → OKtoread.signal

R2 read shared file

readercount := readercount-1;
If readercount = 0 then
OKtowrite.signal;

(R1 finishes)

R2 → readercount := readercount-1;
If readercount = 0 then OKtowrite.signal

busy := true

write shared file

(R2 finishes)

busy := false
If OKtoread.queue then OKtoread.signal
else OKtowrite.signal

Pthread: Mutex

- Provides synchronization to a shared resource
- A typical sequence in the use of a mutex is as follows:
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - Only one succeeds and that thread owns the mutex
  - The owner thread performs some set of actions
  - The owner unlocks the mutex
  - Another thread acquires the mutex
  - … …
  - Finally the mutex is destroyed

POSIX Thread Library
include <pthread.h>

- int **pthread_mutex_init** (pthread_mutex_t *mutex,
  const pthread_mutexattr_t *attr)
  - initializes a *mutex* with the specified attributes
  - 1st argument: the address of the newly created *mutex*
  - 2nd argument: the address to the variable containing the mutex attributes object
    - To use the default attributes, use NULL as the second argument

POSIX Thread Library
include <pthread.h>

- int **pthread_mutex_destroy** (pthread_mutex_t *mutex)
  - Free a mutex object that is no longer needed.

POSIX Thread Library
include <pthread.h>

- int **pthread_mutex_lock** (pthread_mutex_t *mutex)
  - acquires ownership of the *mutex* specified
  - If the specified *mutex* is currently locked, the calling thread is blocked until the *mutex* is available
  - 1st argument: the address of the *mutex* to be locked

- int **pthread_mutex_unlock** (pthread_mutex_t *mutex)
  - unlocks the *mutex* specified
  - 1st argument: the address of the *mutex* to be unlocked

- Important: make sure all threads, that need access to the shared data, use mutex. Otherwise, the shared data could still be corrupted.

Pthread: Condition variable

- A mutex makes other threads to wait while the thread holding the mutex executes code in a critical section.
- A condition variable is typically used by a thread (that has already acquired a mutex) to make itself wait (and temporarily release the mutex) until signaled by another thread.

POSIX Thread Library
include <pthread.h>

- int **pthread_cond_init** (pthread_cond_t *cond,
  const pthread_condattr_t *attr)
  - initializes a condition variable object with the specified attributes
  - 1st argument: the address of the newly created *condition variable*
  - 2nd argument: the address to the variable containing the condition variable attributes object
    - To use the default attributes, use NULL as the second argument

POSIX Thread Library
include <pthread.h>

- int **pthread_cond_wait** (pthread_cond_t *cond,
  pthread_mutex_t *mutex)
  - blocks the calling thread on the condition variable *cond*
    - 1st argument: the address of the condition variable to wait on
    - 2nd argument: the address of the mutex associated with the condition variable
  - when **pthread_cond_wait()** is called, the calling thread must have the associated *mutex* locked
  - the **pthread_cond_wait()** function unlocks the associated *mutex* and blocks on the condition variable (waiting for another thread to signal the condition)

POSIX Thread Library
include <pthread.h>

- int **pthread_cond_signal** (pthread_cond_t *cond)
  - wakes up one thread that is waiting on the condition variable
  - 1st argument: the address of the condition variable
  - the thread that is just waked up automatically requests for the mutex that it unlocked when calling *pthread_cond_wait();* it can resume its execution only after it has re-acquired the *mutex*

Example: Implementing RW-Monitor

```
/*Shared variables*/

int readercount;
int busy;
pthread_mutex_t mutex;
pthread_cond_t OKtoread, OKtowrite;
int QL_OKtoread;
```

Monitor Reader-Writer

begin

readercount: integer

busy: boolean

OKtoread, OKtowrite: condition

Example: Implementing RW-Monitor

```
/*Initialization*/


void init()
{
  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&OKtoread, NULL);
  pthread_cond_init(&OKtowrite, NULL);
  readercount=0;
  busy=0;
  QL_OKtoread=0;
}
```

```
begin /* initialization of local
data */
  readercount := 0;
  busy := false;
end
```

Example: Implementing RW-Monitor

```
/*Procedure startread*/
void startread()
{
  pthread_mutex_lock(&mutex);
    QL_OKtoread++;
    while(busy)
            pthread_cond_wait(
            &OKtoread, &mutex);
    QL_OKtoread--;
    readercount++;
    pthread_cond_signal(&OKtoread);
  pthread_mutex_unlock(&mutex);
}
```

```
procedure startread
begin
   If busy then OKtoread.wait;
   readercount := readercount + 1;
   OKtoread.signal
end startread
```

Example: Implementing RW-Monitor

```
/*Procedure endread*/
void endread()
{
  pthread_mutex_lock(&mutex);
    readercount--;
    if(readercount==0)
     pthread_cond_signal(&OKtowrite);
  pthread_mutex_unlock(&mutex);
}
```

```
procedure endread;
begin
   readercount := readercount-1;
   If readercount = 0 then
     OKtowrite.signal;
end endread
```

Example: Implementing RW-Monitor

```
/*Procedure startwrite*/
void startwrite()
{
  pthread_mutex_lock(&mutex);
    while(busy||readercount!=0)
     pthread_cond_wait(&OKtowrite,
     &mutex);
    busy=1;
  pthread_mutex_unlock(&mutex);
}
```

```
procedure startwrite;
begin
    If busy or readercount <>0 then
     OKtowrite.wait;
    busy := true;
end startwrite
```

Example: Implementing RW-Monitor

```
/*Procedure endwrite*/
void endwrite()
{
  pthread_mutex_lock(&mutex);
    busy=0;
    if(QL_OKtoread>0)
     pthread_cond_signal(&OKtoread);
    else
     pthread_cond_signal(&OKtowrite);
  pthread_mutex_unlock(&mutex);
}
```

```
procedure endwrite;
begin
   busy := false
   if OKtoread.queue then
     OKtoread.signal
   else OKtowrite.signal
end endwrite
```

Example: Using RW-Monitor

```
/*reader*/
void *reader(void *x)
{
startread();
//code to read
endread();
}
```

```
/*writer*/
void *writer(void *x)
{
startwrite();
//code to write
endwrite();
}
```