Lecture 5 Synchronization

I233E OPERATING SYSTEMS

RAZVAN BEURAN

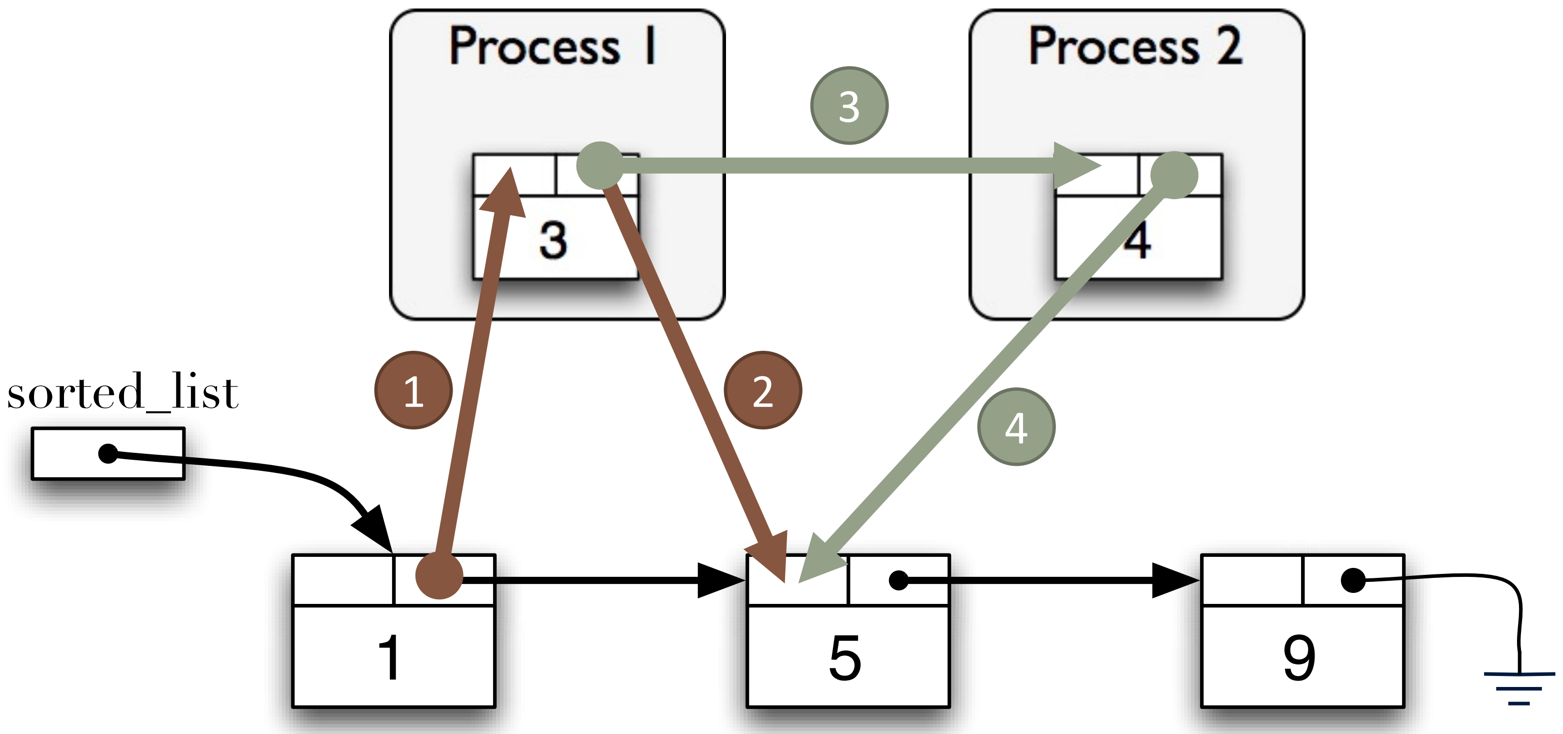
Today's Topics

Synchronization
- Why it is needed

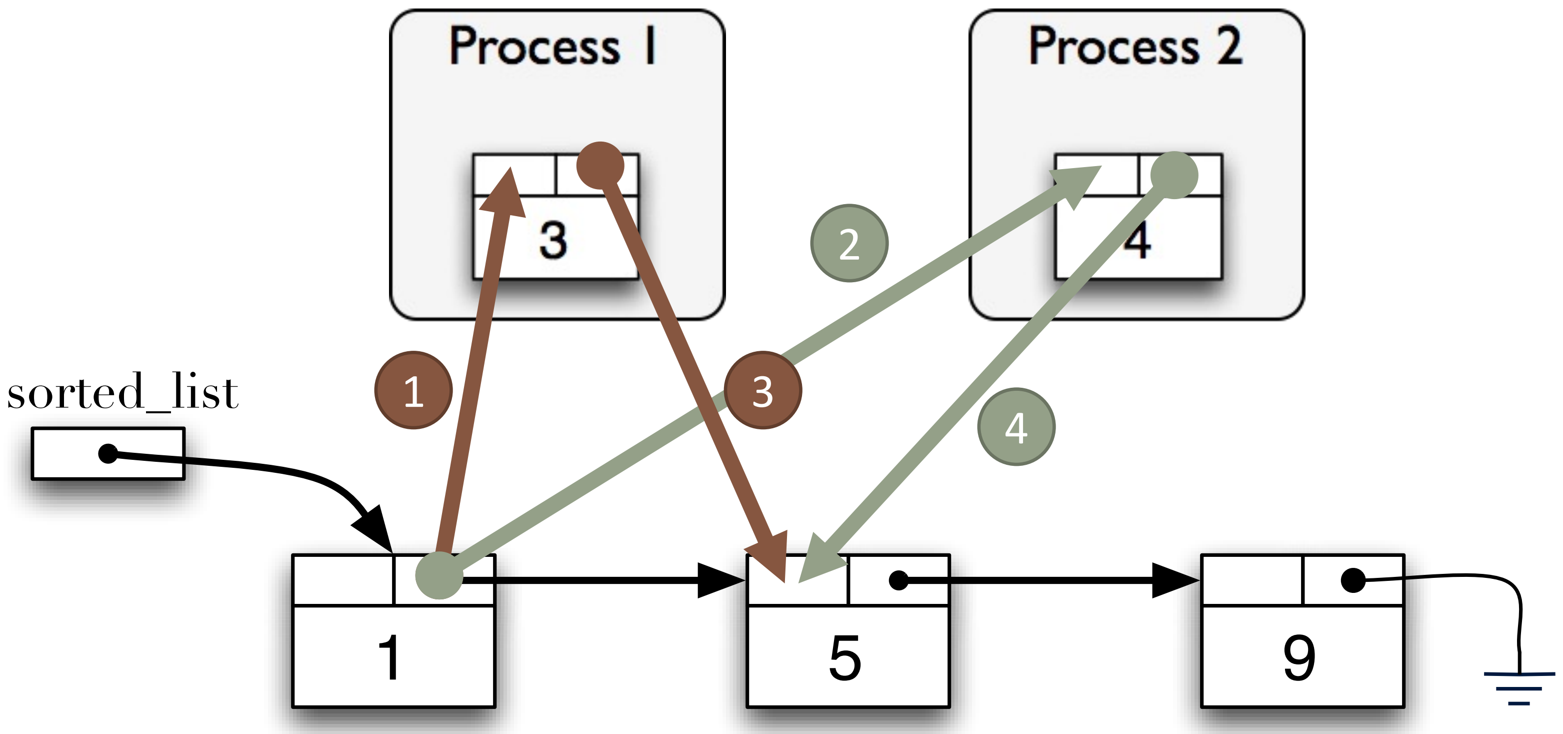
Classical problem #1: Mutual exclusion
- Problem definition
- Solutions

Classical problem #2: Producer-consumer
- Problem definition
- Solutions

Race Condition: Correct Result



Race Condition: Wrong Result



Problem #1
Mutual Exclusion

Mutual Exclusion

```
loop
    ENTER_SECTION()
    critical section;
    LEAVE_SECTION()
    remainder;
end loop
```

Assumptions

- Total of *n* processes
- One critical section

Definition

- No two processes may execute simultaneously inside the critical section
- No process running outside its critical section may block other processes
- No process should wait forever to enter its critical section
- No assumptions are made on speed or number of CPUs

Mechanisms

Disable interrupts

Lock variables (race condition)

Strict alternation

Peterson’s algorithm

Test & set

Compare & swap

Disable Interrupts

Processes (inside kernel)

- *Enter critical section*

```
/* Disable interrupts (assembly) */
CLI;     /* Clear Interrupt Flag (disable) */
```

- ***Critical section***
- *Leave critical section*

```
/* Enable interrupts */
STI;     /* Set Interrupt Flag (enable) */
```

Only works inside the kernel!

Lock Variables (naive)

Mechanism

- Shared variable

```
int lock = 0;
```

Processes

- *Enter critical section*

```
while (lock != 0) {
    /* Wait */
}
lock = 1;
```

Race Condition!

- ***Critical section***
- *Leave critical section*

```
lock = 0;
```

Doesn't *actually* work
(because of race condition)

Strict Alternation

Process 0

- *Enter critical section*

```
while (turn!=0) {
    /* Wait */
}
```

- ***Critical section***
- *Leave critical section*

```
turn = 1;
```

Process 1

- *Enter critical section*

```
while (turn!=1) {
    /* Wait */
}
```

- ***Critical section***
- *Leave critical section*

```
turn = 0;
```

Processes must always take turns!

Peterson's Algorithm

Mechanism

- Shared variables

```
bool interested[2] = {FALSE, FALSE};
int turn;
```

Process 0

- *Enter critical region*

```
interested[0] = TRUE;
turn = 1; /* Give away the turn */
while (interested[1] && turn==1)
   /* Wait */;
```

- ***Critical section***
- *Leave critical region*

```
interested[0] = FALSE;
```

Peterson’s Algorithm (cont.)

Mechanism

- Shared variables

```
bool interested[2] = {FALSE, FALSE};
int turn;
```

Process 1

- *Enter critical region*

```
interested[1] = TRUE;
turn = 0; /* Give away the turn */
while (interested[0] && turn==0)
   /* Wait */;
```

- ***Critical section***
- *Leave critical region*

```
interested[1] = FALSE;
```

Test & Set

Mechanism

- Use the "test and set" function (atomic)

```
atomic bool test_and_set (bool *target) {
  bool value = *target;
  *target = true;
  return value;
}
```

Each process actions

- *Enter critical section*

```
while (test_and_set(&lock))
   /* Wait */;
```

- ***Critical section***
- *Leave critical section*

```
lock = false;
```

Compare & swap

Mechanism

- Use the “cmp_and_swap” function (atomic)

```
atomic int cmp_and_swap(int *val, int expect, int new){
  int temp = *val;
  if (*val == expected) *val = new;
  return temp;
}
```

Each process actions

- Enter critical section

```
while (cmp_and_swap(&lock, 0, 1))
   /* Wait */;
```

- ***Critical section***
- Exit critical section

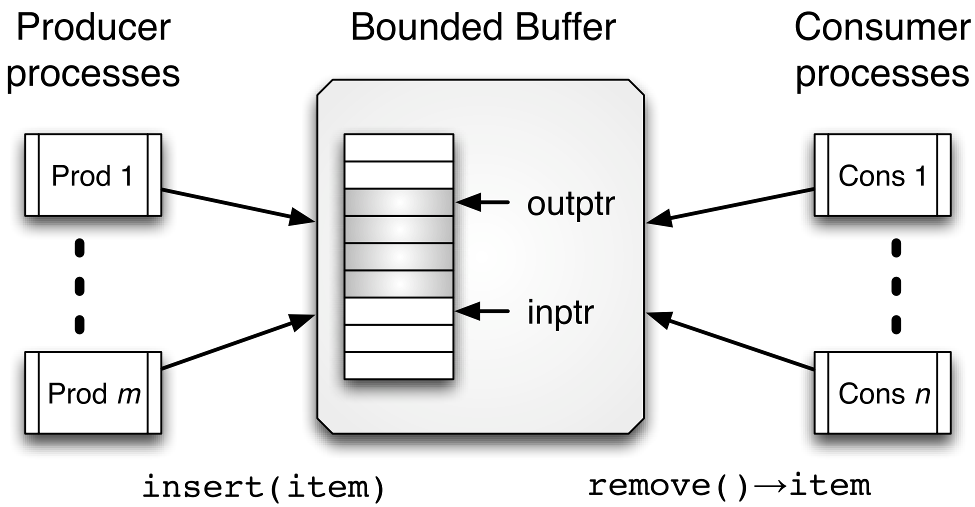
```
lock = 0;
```

Problem #2
Producer Consumer

Producer Consumer

Problem definition

- Fixed size buffer (circular)
- *m* producers; *n* consumers
- Buffer FULL → producer must wait
- Buffer EMPTY → consumer must wait



Producer Consumer (cont.)

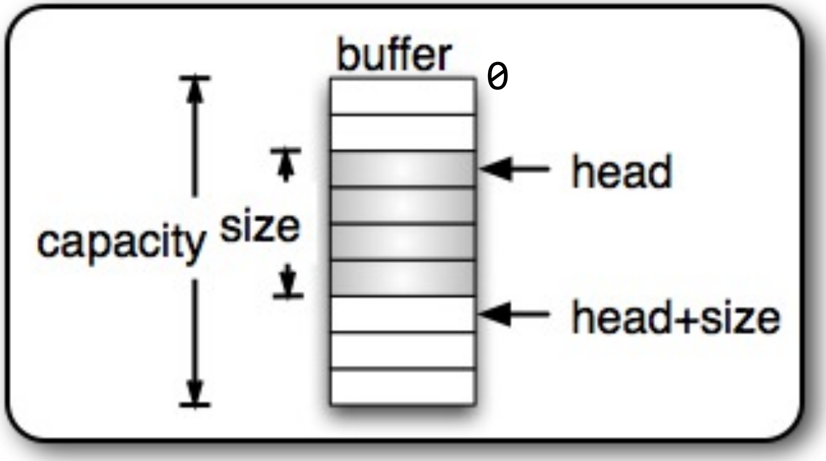
Operations

- void **INSERT**(item)

put()
```
// ???
buffer[(head+size)%capacity] = item;
size++;
// ???
```

- item **REMOVE**()

get()
```
// ???
item = buffer[head];
head = (head+1)%capacity;
size--;
// ???
return item;
```



Mechanisms

Semaphore

Mutex & conditions

Monitor

Semaphore

Initialization

- Initial value of the semaphore represents the number of available “resources”

```
semaphore sem = <init value>;
```

Operations (atomic)

- **Down** (sem)

```
sem--;
if (sem < 0)
  sleep;
```

- **Up** (sem)

```
if (sem < 0)
  Wake up one thread;
sem++;
```

Producer-Consumer w/ Semaphores

Initialization

```
semaphore mutex = 1;
semaphore empty = capacity; // number of empty slots
semaphore full  = 0;        // number of full slots
```

void INSERT(item)

Critical section

```
Down(&empty);
Down(&mutex);
put(item);
Up(&mutex);
Up(&full);
```

item REMOVE()

Critical section

```
Down(&full);
Down(&mutex);
item = get();
Up(&mutex);
Up(&empty);
return item;
```

Pthread: Mutex

Variables

- Mutex type

```
pthread_mutex_t mutex;
```

Operations

- Initialize mutex

```
pthread_mutex_init(&mutex, NULL);
```

- Lock/unlock/try to lock mutex

```
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
res = pthread_mutex_trylock(&mutex);
```

- Destroy mutex

```
pthread_mutex_destroy(&mutex);
```

Pthread: Conditions

Variables

- Condition variable type

```
pthread_cond_t cond;
```

Operations

- Initialize condition

```
pthread_cond_init(&cond, NULL);
```

- Wait for condition

```
pthread_cond_wait(&cond, &mutex);
pthread_cond_timedwait(&cond, &mutex, &timeout);
```

- Unblock thread(s) waiting for condition

```
pthread_cond_signal(&cond);
```

- Destroy condition

```
pthread_cond_destroy(&cond);
```

Details on cond_wait

Assumption

- Called with mutex locked by the calling thread

What it does atomically

- Releases mutex **AND**
- Blocks thread until condition is next signaled via the condition signal function (or some interruption occurs)
  - Past signals are not "queued"!

When the function returns

- Mutex is already locked again

Producer-Consumer w/ Mutex & Conditions

Initialization

```
mutex_t mutex;
cond_t  nfull, nempty; // not full, not empty

//initialization code...
```

void INSERT(item)

Critical section

```
mutex_lock(&mutex);
if (size == capacity)
  cond_wait(&nfull, &mutex);
put(item); //increments size!
cond_signal(&nempty);
mutex_unlock(&mutex);
```

item REMOVE()

```
mutex_lock(&mutex);
if (size == 0)
  cond_wait(&nempty, &mutex);
item = get(); //decrements size!
cond_signal(&nfull);
mutex_unlock(&mutex);
return item;
```

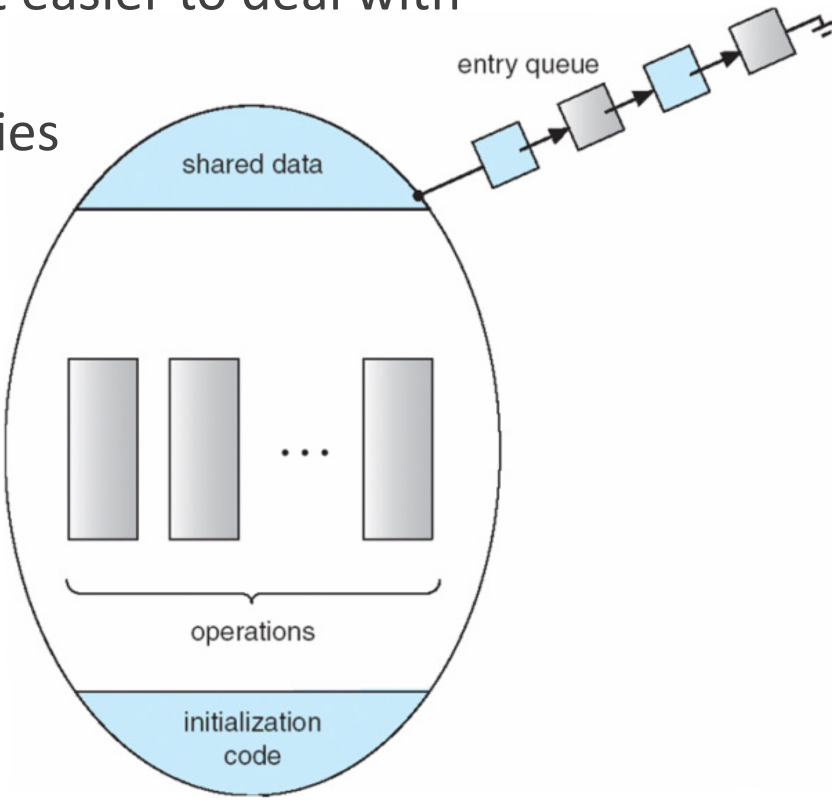
Monitor

Description

- Programming construct to make it easier to deal with synchronization
- Module with set of methods/entries
- Encapsulates mutual exclusion mechanisms

Concurrency control

- Implicit mutex for the included operations
- Condition variables for testing conditions



Producer-Consumer w/ Monitor

Pseudo-Pascal syntax

```
monitor producer_consumer
 condition nfull, nempty;   // not full, not empty
 var size : integer init 0; // number of items in buffer
 ...

procedure insert(item: integer)
 begin
  if size = capacity then
     nfull.wait;
  put(item);
  nempty.signal;
 end;

end monitor;
```

```
function remove : integer
begin
 if size = 0 then
    nempty.wait;
 item = get();
 nfull.signal;
 return item;
end;
```

Summary

Synchronization

- Needed to prevent race conditions

Mutual exclusion

- Control access to a critical section
- Support provided by OS/hardware

Producer-Consumer

- Various implementations possible
  - Semaphores
  - Mutex & conditions

Next Time

Deadlocks

Deadlock avoidance

Deadlock detection

NOTE

- Assignment #2
  - Will be uploaded to JAIST-LMS on Oct. 29
  - Due on Nov. 4 at 23:59
  - Solution during tutorial hour on Nov. 5