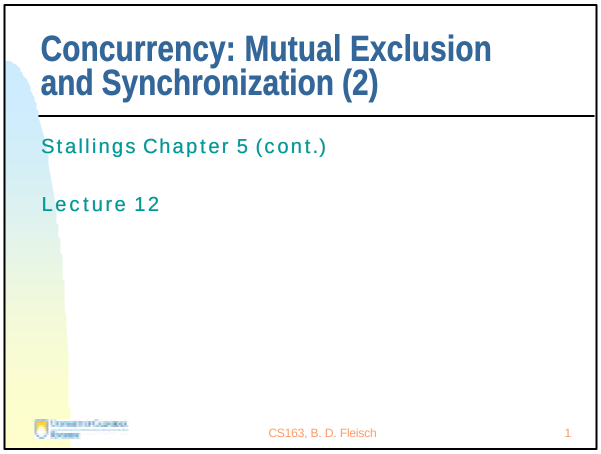
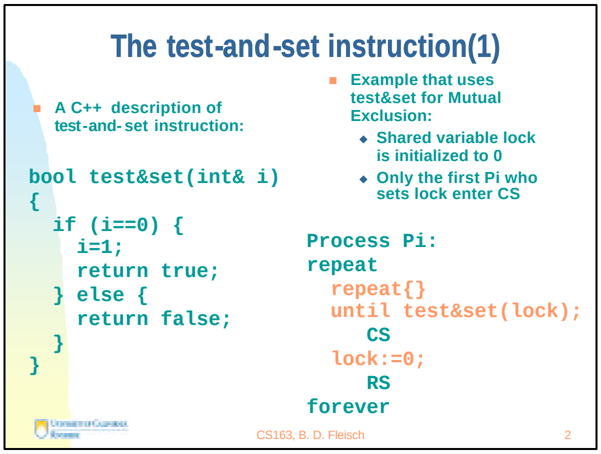


Concurrency: Mutual Exclusion and Synchronization (2)

Stallings Chapter 5 (cont.)

Lecture 12



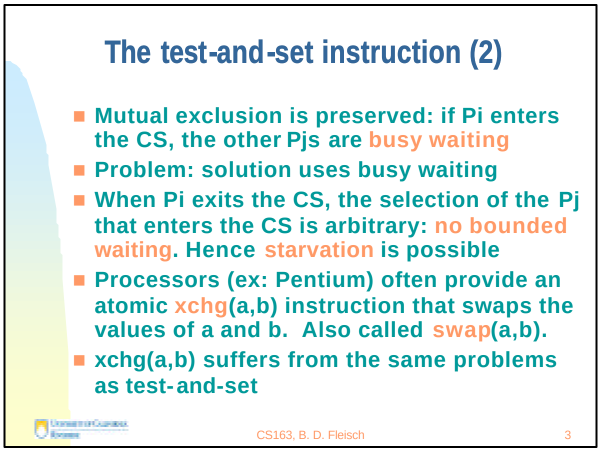
The test-and-set instruction(1)

- A C++ description of test-and-set instruction:

```
bool test&set(int& i)
{
  if (i==0) {
    i=1;
    return true;
  } else {
    return false;
  }
}
```

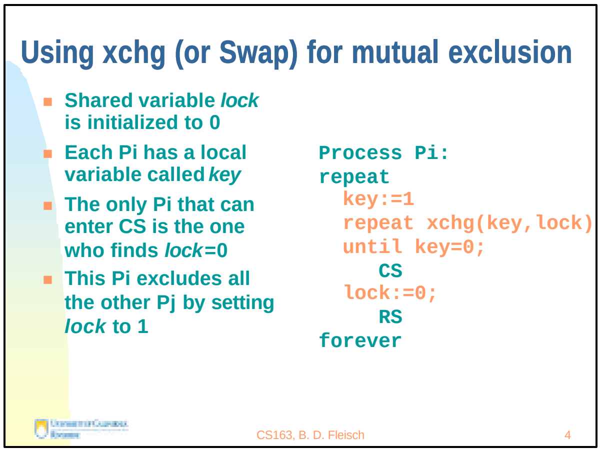
- Example that uses test&set for Mutual Exclusion:
  - Shared variable lock is initialized to 0
  - Only the first Pi who sets lock enter CS

```
Process Pi:
repeat
  repeat{}
  until test&set(lock);
     CS
  lock:=0;
     RS
forever
```



The test-and-set instruction (2)

- Mutual exclusion is preserved: if Pi enters the CS, the other Pjs are busy waiting
- Problem: solution uses busy waiting
- When Pi exits the CS, the selection of the Pj that enters the CS is arbitrary: no bounded waiting. Hence starvation is possible
- Processors (ex: Pentium) often provide an atomic xchg(a,b) instruction that swaps the values of a and b. Also called swap(a,b).
- xchg(a,b) suffers from the same problems as test-and-set



Using xchg (or Swap) for mutual exclusion

- Shared variable *lock* is initialized to 0
- Each Pi has a local variable called *key*
- The only Pi that can enter CS is the one who finds *lock*=0
- This Pi excludes all the other Pj by setting *lock* to 1

```
Process Pi:
repeat
  key:=1
  repeat xchg(key,lock)
  until key=0;
     CS
  lock:=0;
     RS
forever
```

Software solutions

- We consider first the case of 2 process solutions
  - Algorithm 1 - 3 are incorrect
  - Algorithm 4 is correct (Peterson's algorithm)
- Then we generalize to n processes
  - Lamport's Bakery algorithm
- Notation
  - We have 2 processes: P0 and P1
  - When presenting process Pi, Pj always denote the other process (i != j)

Algorithm 1

- The shared variable turn is initialized (to 0 or 1) before executing any P_i
- Pi's critical section is executed iff turn = i
- Pi is busy waiting if P_j is in CS: mutual exclusion is satisfied
- Progress requirement is not satisfied since it requires strict alternation of CSs

```
Process Pi:
repeat
  while(turn!=i){};
     CS
  turn:=j;
     RS
forever
```

Algorithm 2

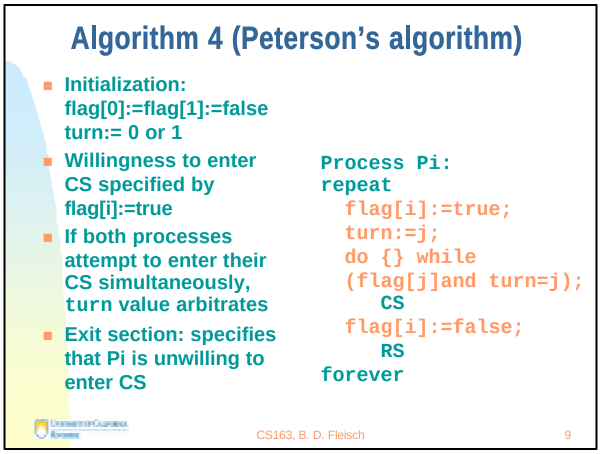
- Keep a Boolean variable for each process: flag[0] and flag[1]
- Pi signals that it is ready to enter it's CS by: flag[i]:=true
- First check flag[] other process before proceeding
- Does not satisfy correctness requirement

```
Process Pi:
repeat
  while(flag[j]){};
  flag[i]:=true;
     CS
  flag[i]:=false;
     RS
forever
```

Algorithm 3

- Keep a Boolean variable for each process: flag[0] and flag[1]
- Pi signals that it is ready to enter it's CS by: flag[i]:=true
- ME is satisfied but not the progress requirement
- If we have the sequence:
  - T0: flag[0]:=true
  - T1: flag[1]:=true
- Both process will wait forever to enter their CS: we have a deadlock

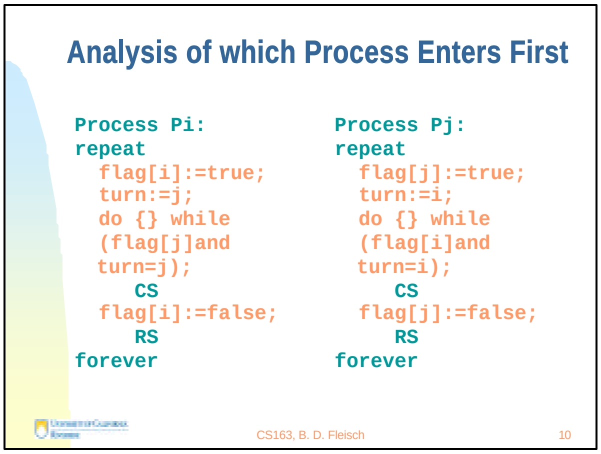
```
Process Pi:
repeat
  flag[i]:=true;
  while(flag[j]){};
     CS
  flag[i]:=false;
     RS
forever
```



Algorithm 4 (Peterson's algorithm)

- Initialization:
  flag[0]:=flag[1]:=false
  turn:= 0 or 1
- Willingness to enter CS specified by flag[i]:=true
- If both processes attempt to enter their CS simultaneously, `turn` value arbitrates
- Exit section: specifies that Pi is unwilling to enter CS

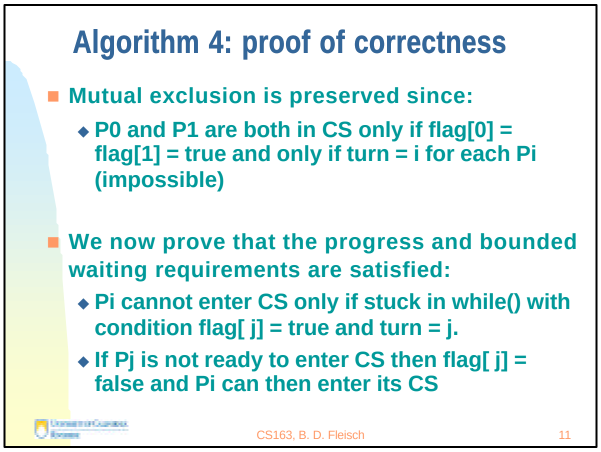
```
Process Pi:
repeat
  flag[i]:=true;
  turn:=j;
  do {} while
  (flag[j]and turn=j);
     CS
  flag[i]:=false;
     RS
forever
```



Analysis of which Process Enters First

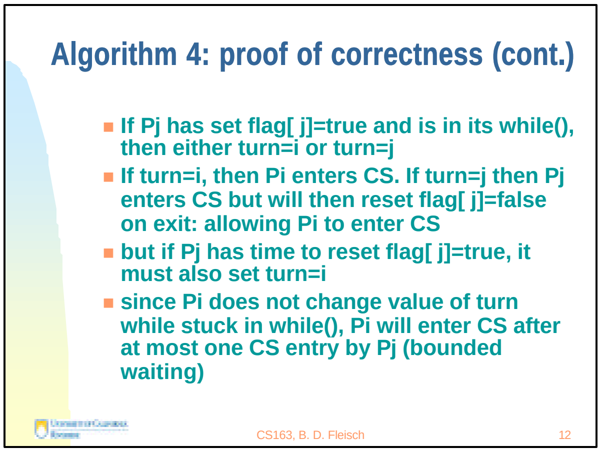
```
Process Pi:
repeat
  flag[i]:=true;
  turn:=j;
  do {} while
  (flag[j]and
  turn=j);
     CS
  flag[i]:=false;
     RS
forever
```

```
Process Pj:
repeat
  flag[j]:=true;
  turn:=i;
  do {} while
  (flag[i]and
  turn=i);
     CS
  flag[j]:=false;
     RS
forever
```



Algorithm 4: proof of correctness

- Mutual exclusion is preserved since:
  - P0 and P1 are both in CS only if flag[0] = flag[1] = true and only if turn = i for each Pi (impossible)
- We now prove that the progress and bounded waiting requirements are satisfied:
  - Pi cannot enter CS only if stuck in while() with condition flag[j] = true and turn = j.
  - If Pj is not ready to enter CS then flag[j] = false and Pi can then enter its CS



Algorithm 4: proof of correctness (cont.)

- If Pj has set flag[j]=true and is in its while(), then either turn=i or turn=j
- If turn=i, then Pi enters CS. If turn=j then Pj enters CS but will then reset flag[j]=false on exit: allowing Pi to enter CS
- but if Pj has time to reset flag[j]=true, it must also set turn=i
- since Pi does not change value of turn while stuck in while(), Pi will enter CS after at most one CS entry by Pj (bounded waiting)

What about process failures?

- If all 3 criteria (ME, progress, bounded waiting) are satisfied, then a valid solution will provide robustness against failure of a process in its remainder section (RS)
  - since failure in RS is just like having an infinitely long RS
- However, no valid solution can provide robustness against a process failing in its critical section (CS)
  - A process Pi that fails in its CS does not signal that fact to other processes: for the others Pi is still in its CS