

Synchronization in Practice

- User program synchronization
 - for threads
 - for processes
- OS kernel synchronization

10/20/2010

CSC 2/456

34

User Program Synchronization for Processes

- Processes naturally do not share the same address space
- Process synchronization:
 - semaphore
 - shared memory
 - pipes

10/20/2010

CSC 2/456

35

User Program Synchronization for Threads

- All threads share the same address space
- When only need to protect a short critical section (busy waiting is OK)
 - software/hardware spin locks
 - still has the risk of context switch in the middle of critical section
- For complex synchronization (busy waiting is not OK)
 - semaphore, mutex lock, condition variable, ...
 - may need kernel help
- In pthreads
 - mutex lock and condition variable
 - condition variable must be used together with a mutex lock

10/20/2010

CSC 2/456

36

Synchronization Primitives in Pthreads

- Mutex lock
 - pthread_mutex_init
 - pthread_mutex_destroy
 - pthread_mutex_lock
 - pthread_mutex_unlock
- Condition variable (used in conjunction with a mutex lock)
 - pthread_cond_init
 - pthread_cond_destroy
 - pthread_cond_wait
 - pthread_cond_signal
 - pthread_cond_broadcast

10/20/2010

CSC 2/456

37

Mutex Locks: Creation and Destruction

```
pthread_mutex_init(  
    pthread_mutex_t * mutex,  
    const pthread_mutex_attr *attr);
```

- Creates a new mutex lock
- ```
pthread_mutex_destroy(
 pthread_mutex_t *mutex);
```
- Destroys the mutex specified by mutex

10/20/2010

CSC 2/456

38

## Mutex Locks: Lock

```
pthread_mutex_lock(
 pthread_mutex_t *mutex)
```

- Tries to acquire the lock specified by mutex.
- If mutex is already locked, then calling thread blocks until mutex is unlocked.

10/20/2010

CSC 2/456

39

## Mutex Locks: UnLock

```
pthread_mutex_unlock(
 pthread_mutex_t *mutex);
```

- If calling thread has mutex currently locked, this will unlock the mutex.
- If other threads are blocked waiting on this mutex, one will unblock and acquire mutex
- Which one is determined by the scheduler

10/20/2010

CSC 2/456

40

## Condition variables: Creation and Destruction

```
pthread_cond_init(
 pthread_cond_t *cond,
 pthread_cond_attr *attr)
```

- Creates a new condition variable cond
- ```
pthread_cond_destroy(  
    pthread_cond_t *cond)
```
- Destroys the condition variable cond.

10/20/2010

CSC 2/456

41

Condition Variables: Wait

```
pthread_cond_wait(  
    pthread_cond_t *cond,  
    pthread_mutex_t *mutex)
```

- Blocks the calling thread, waiting on cond
- Unlocks the mutex
- Re-acquires the mutex when unblocked

10/20/2010

CSC 2/456

42

Condition Variables: Signal

```
pthread_cond_signal(  
    pthread_cond_t *cond)
```

- Unlocks one thread waiting on cond.
- Which one is determined by scheduler.
- If no thread waiting, then signal is a no-op.

10/20/2010

CSC 2/456

43

Condition Variables: Broadcast

```
pthread_cond_broadcast(  
    pthread_cond_t *cond)
```

- Unlocks all threads waiting on cond.
- If no thread waiting, then broadcast is a no-op.

10/20/2010

CSC 2/456

44

Use of Condition Variables

- **IMPORTANT NOTE:** A signal is “forgotten” if there is no corresponding wait that has already occurred
- Use semaphores (or construct a semaphore) if you want the signal to be remembered

10/20/2010

CSC 2/456

45

TSP (Traveling Salesman)

- Goal:
 - given a list of cities, a matrix of distances between them, and a starting city,
 - find the shortest tour in which all cities are visited exactly once.
- Example of an NP-hard search problem.
- Algorithm: branch-and-bound.

10/20/2010

CSC 2/456

46

Branching

- Initialization:
 - go from starting city to each of remaining cities
 - put resulting partial path into priority queue, ordered by its current length.
- Further (repeatedly):
 - take head element out of priority queue,
 - expand by each one of remaining cities,
 - put resulting partial path into priority queue.

10/20/2010

CSC 2/456

47

Finding the Solution

- Eventually, a complete path will be found.
- Remember its length as the current shortest path.
- Every time a complete path is found, check if we need to update current best path.
- When priority queue becomes empty, best path is found.

10/20/2010

CSC 2/456

48

Using a Simple Bound

- Once a complete path is found, we have a lower bound on the length of shortest path
- No use in exploring partial path that is already longer than the current lower bound
- Better bounding methods exist ...

10/20/2010

CSC 2/456

49

Sequential TSP: Data Structures

- Priority queue of partial paths.
- Current best solution and its length.
- For simplicity, we will ignore bounding.

10/20/2010

CSC 2/456

50

Sequential TSP: Code Outline

```

init_q(); init_best();
while( (p=de_queue()) != NULL ) {
  for each expansion by one city {
    q = add_city(p);
    if( complete(q) ) { update_best(q) };
    else { en_queue(q) };
  }
}

```

10/20/2010

CSC 2/456

51

Parallel TSP: Possibilities

- Have each process do one expansion
- Have each process do expansion of one partial path
- Have each process do expansion of multiple partial paths
- Issue of granularity/performance, not an issue of correctness.
- Assume: process expands one partial path.

10/20/2010

CSC 2/456

52

Parallel TSP: First Cut (part 1)

```

process i:
while( (p=de_queue()) != NULL ) {
  for each expansion by one city {
    q = add_city(p);
    if complete(q) { update_best(q) };
    else en_queue(q);
  }
}

```

10/20/2010

CSC 2/456

53

Parallel TSP: First cut (part 2)

- In de_queue: wait if q is empty
- In en_queue: signal that q is no longer empty

10/20/2010

CSC 2/456

54

Parallel TSP

```

process i:
  while( (p=de_queue()) != NULL ) {
    for each expansion by one city {
      q = add_city(p);
      if complete(q) { update_best(q) };
      else en_queue(q);
    }
  }

```

10/20/2010

CSC 2/456

55

Parallel TSP: Critical Sections

- All concurrently accessed shared data must be protected by critical section
- Update_best must be protected by a critical section
- En_queue and de_queue must be protected by the same critical section

10/20/2010

CSC 2/456

56

Termination condition

- How do we know when we are done?
- All processes are waiting inside de_queue.
- Count the number of waiting processes before waiting.
- If equal to total number of processes, we are done.

10/20/2010

CSC 2/456

57

Parallel TSP: Mutual Exclusion

```

en_queue() / de_queue() {
    pthread_mutex_lock(&queue);
    ...;
    pthread_mutex_unlock(&queue);
}
update_best() {
    pthread_mutex_lock(&best);
    ...;
    pthread_mutex_unlock(&best);
}
    
```

10/20/2010

CSC 2/456

58

Parallel TSP: Condition Synchronization

```

de_queue() {
    pthread_mutex_lock(&queue);
    while( (q is empty) and (not done) ) {
        waiting++;
        if( waiting == p ) {
            done = true;
            pthread_cond_broadcast(&empty);
        }
        else {
            pthread_cond_wait(&empty, &queue);
            waiting--;
        }
    }
    if( done )
        return null;
    else
        remove and return head of the queue;
    pthread_mutex_unlock(&queue);
}
    
```

10/20/2010

CSC 2/456

59

DEADLOCK

10/20/2010

CSC 2/456

60

The Deadlock Problem

- Definition:
 - A set of blocked processes each holding some resources and waiting to acquire a resource held by another process in the set
 - None of the processes can proceed or back-off (release resources it owns)
- Examples:
 - Dining philosopher problem
 - System has 2 memory pages (unit of memory allocation); P_1 and P_2 each hold one page and each needs another one
 - Semaphores A and B , initialized to 1

P_1	P_2
<i>wait (A);</i>	<i>wait(B)</i>
<i>wait (B);</i>	<i>wait(A)</i>

10/20/2010

CSC 2/456

61

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously:

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n, P_0\}$ of waiting processes such that
 - P_0 is waiting for a resource that is held by P_1 ,
 - P_1 is waiting for a resource that is held by P_2 ,
 - ...,
 - P_{n-1} is waiting for a resource that is held by P_n ,
 - and P_n is waiting for a resource that is held by P_0 .

10/20/2010

CSC 2/456

62

Methods for Handling Deadlocks

- Ignore the problem and pretend that deadlocks would never occur
- Ensure that the system will *never* enter a deadlock state (prevention or avoidance)
- Allow the system to enter a deadlock state and then detect/recover

10/20/2010

CSC 2/456

63

The Ostrich Algorithm

- Pretend there is no problem
 - unfortunately they can occur
- Reasonable if
 - deadlocks occur very rarely
 - cost of prevention is high
- Your typical OSes take this approach
- It is a trade off between
 - convenience
 - correctness

10/20/2010

CSC 2/456

64

Deadlock Prevention

Restrain the ways requests can be made to break one of the four necessary conditions for deadlocks.

Attacking the Mutual Exclusion Condition:

- Some devices (such as printer) can be spooled
 - only the printer daemon uses printer resource
 - thus deadlock for printer eliminated
- Not all devices can be spooled

10/20/2010

CSC 2/456

65

Deadlock Prevention

Attacking the Hold and Wait Condition:

- Require processes to request all resources before starting
- Problems
 - may not know required resources at start of run
 - also ties up resources other processes could be using
- Variation:
 - before a process requests a new resource, it must give up all resources and then request all resources needed

10/20/2010

CSC 2/456

66

Deadlock Prevention

Attacking the No Preemption Condition:

- Preemption
 - when a process is holding some resources and waiting for others, its resources may be preempted to be used by others
- Problem
 - Many resources may not allow preemption; i.e., preemption will cause process to fail

Attacking the Circular Wait Condition:

- impose a total order of all resource types; and require that all processes request resources in the same order

10/20/2010

CSC 2/456

67

Deadlock Avoidance

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state*
- System is in safe state if there exists a safe sequence of all processes
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is safe if for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$.
 - If P_i 's resource needs are not immediately available, then P_i can wait until all P_j have finished
 - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

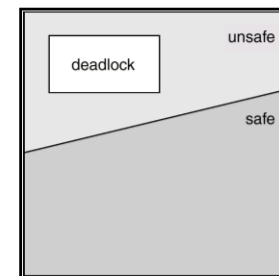
10/20/2010

CSC 2/456

68

Deadlock Avoidance (cont.)

- If a system is in safe state \Rightarrow no deadlocks
- If a system is in unsafe state \Rightarrow possibility of deadlock
- Deadlock avoidance
 - dynamically examines the resource-allocation state
 - ensures that a system will never enter an unsafe state



10/20/2010

CSC 2/456

69

Banker's Algorithm

- Each process must a priori claim the maximum set of resources that might be needed in its execution
- Safety check
 - repeat
 - pick any process that can finish with existing available resources; finish it and release all its resources
 - until no such process exists
 - all finished → safe; otherwise → unsafe.
- When a resource request is made, the process must wait if:
 - enough available resource is not available for this request
 - granting the request would result in an unsafe system state

10/20/2010

CSC 2/456

70

Example of Banker's Algorithm

- 5 processes P_0 through P_4
- 3 resource types: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T_0 :

	<u>Allocation</u>			<u>MaxNeeds</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

- Is this a safe state?
- Can request for (1,0,2) by P_1 be granted?

10/20/2010

CSC 2/456

71

Methods for Handling Deadlocks

- Ignore the problem and pretend that deadlocks would never occur
- Ensure that the system will *never* enter a deadlock state
- Allow the system to enter a deadlock state and then detect/recover

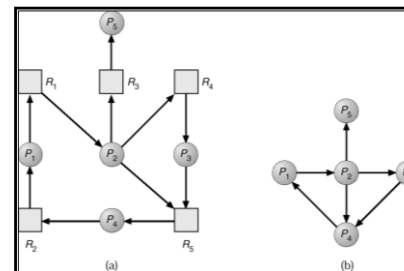
10/20/2010

CSC 2/456

72

Single Instance of Each Resource Type

- Maintain *wait-for* graph
 - Nodes are processes.
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j .
- Periodically search for a cycle in the graph.



10/20/2010

CSC 2/456

73

Additional Issues

- When there are several instances of a resource type
 - cycle detection in wait-for graph is not sufficient
- Deadlock detection is very similar to the safety check in the Banker's algorithm
 - just replace the maximum needs with the current requests

10/20/2010

CSC 2/456

74

Recovery from Deadlock

- Recovery through preemption
 - take a resource from some other process
 - depends on nature of the resource
- Recovery through rollback
 - checkpoint a process state periodically
 - rollback a process to its checkpoint state if it is found deadlocked
- Recovery through killing processes
 - kill one or more of the processes in the deadlock cycle
 - the other processes get its resources
- In which order should we choose process to kill?

10/20/2010

CSC 2/456

75

Disclaimer

- Parts of the lecture slides contain original work of Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, Willy Zwaenepoel, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).

10/20/2010

CSC 2/456

76