More on Synchronization and Deadlock

CS 256/456

Dept. of Computer Science, University of Rochester

Examples of OS Kernel Synchronization

- Two processes making system calls to read/write on the same file, leading to possible race condition on the file system data structures in OS.
- Interrupt handlers put I/O data into a buffer queue that might be retrieved by application-initiated I/O system calls.

OS Kernel Structure for Synchronization

- OS is divided into two parts:
  - upper part (serving application requests): system call, exception
  - lower part (serving hardware device requests): interrupt handling
- Upper part runs in process/thread context
  - resource accounting to corresponding process/thread
  - running on a kernel stack usually associated with the corresponding process/thread control block
- Lower part runs in a separate interrupt context
  - resource accounting to who?
  - running in a separate (often dedicated) kernel interrupt stack
- Blocking behaviors:
  - Upper part may block (yield CPU), interleave with others.
  - Lower part does not block, must run atomically (one by one) – interrupt handlers typically run with other interrupts disabled. Why?
- Preemption/priority:
  - A lower part interrupt handler may preempt an upper part system call processing, not vice versa.

OS Kernel Synchronization

- Available mechanisms:
  - disabling interrupts
  - spin_lock (busy waiting lock)
  - blocking synchronization (mutex lock, semaphore, …)
- Synchronization between upper part kernel "threads"
  - typically blocking synchronization
  - spin_lock if critical section short (only useful on multiprocessor)
- Synchronization between a upper part kernel "thread" and a lower part interrupt handler:
  - if blocking synchronization: block only at upper part, never lower part (possible in semaphore)
  - spin_lock may be used (only useful on multiprocessor)
  - the upper part should disable interrupt before entering critical section

A Little More on OS Kernel Structure

- Lower part interrupt handlers do not block
  - interrupt handlers typically run with other interrupts disabled.
- This can be a problem when interrupt handlers do more and more work
- In modern OS, interrupt handlers typically defer some work to later (interruptible contexts)
  - software irqs in Linux

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
wait (A);	*wait(B)*
wait (B);	*wait(A)*

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 $\{P_0, P_1, \ldots, P_n, P_0\}$ 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.

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.

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

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

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 for a new resource, it must give up all resources and then request all resources needed

Deadlock Prevention

Attacking the No Preemption Condition:

- Preemption
  - when a process 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:

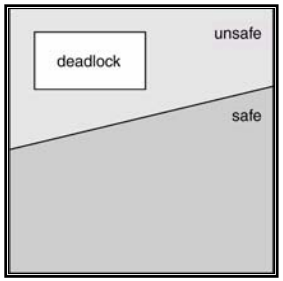
- impose a total order of all resource types; and require that all processes request resources in the same order

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 $<P_1, P_2, \ldots, P_n>$ 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 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.

Deadlock Avoidance (cont.)

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



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:
  - no enough available resource this request
  - granting of such request would result in a unsafe system state

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:

	Allocation	*MaxNeeds*	*Available*
	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	

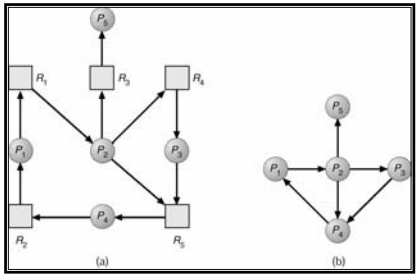
- Is this a safe state?
- Can request for (1,0,2) by P_1 be granted?

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.

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.



Resource-Allocation Graph Corresponding wait-for graph

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

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?

Disclaimer

- Parts of the lecture slides contain original work of Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, 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).