Seongwoo Choi
CMPS 111
Winter 2017
Prof. Darrell Long
Week 5
Notes

February 6th 2017 - February 8th 2017

Critical sections using semaphores

Define a class called Semaphore

- Class allows more complex implementations for semaphores
- Details hidden from processes

Code for individual process is simple.

Implementing semaphores with blocking

Assume two (existing) operations:
sleep (): suspends current process
Wakeup (): allows process P to resume execution

Semaphore is a class

Track value of semaphore
Keep a list of processes waiting for the semaphore
Operations still atomic

```
class Semaphore () {
    int value;
    ProcessList p1;
    void down ();
    Void up ();
};
```

Semaphores for general synchronization

We want to execute B in P1 only after A executes in P0
Use a semaphore initialized to 0
Use up() to notify P1 at the appropriate time
This is called a rendezvous.

Types of semaphores

Two different types of semaphores
Counting semaphores
Binary semaphores
Counting semaphore
Vale can range over an unrestricted range

Binary semaphore

Only two values possible
Value 1 means the semaphore is available
Value 0 means a process has acquired the semaphore
Maybe simpler to implement
Possible to implement one type using the other (midterm)

Deadly Embrace
Dijkstra was romantic. He did not write codes. He wrote only proofs.

Monitors

A monitor is another kind of high-level synchronization primitive
One monitor has multiple entry points
Only one process may be in the monitor at any time
Enforces mutual exclusion: better at avoiding programming errors

Monitors provided by high-level language

Variables belonging to monitor are protected from simultaneous access
Procedures in monitor are guaranteed to have mutual exclusion

Monitor implementation

Language/compiler handles implementation
Can be implemented using semaphores

Xerox invented ethernet, video display, and mouse and Apple stole them

Monitor Usage

This looks like C++ code, but it's not supported by C++
Provides the following features:
Variables foo, bar, and arr are accessible only by func1 & func2
Only one process can be executing in either func1 or func2 at any time

```
monitor mon {
        Int foo;
        Int bar;
        Double arr[100];
        Void func1(_) {
        }
        Void func2(_) {
        }
        Void mon() {  // initialization code
        }
```

}

Set up process called semaphore process and give variables. I send a message and counter is bigger than 0 it executes something, but if it is zero, then wait.

If you have a monitor, then you do not want to use a monitor.

Monitor semantics

Problem: P signals on condition variable X in monitor M, waking Q
Both can't be active in the monitor at the same time
Which one continues first?

Mesa semantics

Signaling process (P) continues first
Q resumes when P leaves the monitor
Seems more logical: why suspend P when it signals?

Hoare semantics

Awakened process (Q) continues first
P resumes when Q leaves the monitor
May be better: condition that Q wanted may no longer hold when P leaves the monitor

Locks & condition variables

Monitors require native language support
Instead, provide monitor support using special

Implementing locks with semaphores

Use mutex to ensure exclusion within the lock bounds
Use next

Peterson's algorithm was mentioned during the class.
Switch the order of something. Everything works

Message passing

Synchronize by exchanging messages
Two primitives:
Send: send a message
Receive: receive a message
Both may specify a "channel" to use
Issue: how does the sender know the receiver got the message?
Issue: authentication.

Barriers

Used for synchronizing multiple processes
Processes wait at a “barrier” until all in the group arrive
After all have arrived, all processes can proceed
May be implemented using locks and condition variables.

Implementing barriers using semaphores

Barrier b;
B.bsem.value

Deadlock and starvation

Deadlock: two or more processes are waiting indefinitely for an event that can only by caused by a waiting process
P0 gets A, needs B
P1 gets B, needs A
Each process waiting for the other to signal

Starvation: indefinite blocking

Process is never removed from the semaphore queue in which its suspended
May be caused by ordering in queues (priority)

Livelock

Sometimes, processes can still run, but not make progress
Example; two processes want to use resources A and B
P0 gets A, P1 gets B
Each realized that a deadlock will occur if then proceed as planned
P0 drops A, P1 drops B
Same problem as before
This can go on for a very long time
Real-world example: Ethernet transmission collisions
If there’s a “collision” on the wire, wait and try again
Multiple processes waited the exact same amount of time…

Classical synchronization problems

Bounded buffer problem
Goal: implement producer-consumer without busy waiting

Readers-writers problem
Shared variables
Int nreaders;
Semaphores mutex(1), writing(1);

Reader process

```
mutex.down();
Nreaders += 1;
If (nreaders == 1) // wait if
   Writing.down(); // 1st reader
mutex.up();
// Read some stuff
mutex.down()
```

Dining Philosophers

N philosophers around a table
All are hungry
All like to think
N chopsticks available
1 between each pair of philosophers
Philosophers need two chopsticks to eat
Philosophers alternate between eating and thinking
Goal: coordinate use of chopsticks

Solution 1
Use a semaphore for each chopstick
A hungry philosopher
Gets the chopstick to his right
Gets

Solution 2
Gets lower, then higher numbered chopstick
Eats
Puts down the chopsticks
Potential problems?
Deadlock
Fairness

Shared variables

```
Const int n;
// initialize to 1
Semaphore chopstick[n];
```

Code for philosopher

```
Int i1, i2;
While (1) {
        If (i != (n-1)) {
        I1
```

Dining philosophers with locks

Next time, we will talk about memory. Virtual memory

Memory

Deadlocks

But crises and deadlocks when they occur have at least this advantage, that they force us to think.

Jawaharlal Nehru

Resources

-> Resource: something a process uses
Usually limited (at least somewhat)
-> Examples of computer resources
Printers
Semaphores / locks
Memory
Tables (in a database)
-> Processes need access to resources in reasonable order
-> Two types of resources:
Preemptable resources: can be taken away from a process with no ill effects
Non Preemptable resources: will cause the process to fail if taken away

Still millions of dollars are used in manufacturing tapes
The property of tape is sequential.

Using resources

-> Sequence of events required to use a resource
    Request the resource
    Use the resource
    Release the resource
-> Can't use the resource if request is denied
    Requesting process has options
        Block and wait for resource
        Continue (if possible) without it: may be able to use an alternate resource
        Process fails with error code
Some of these may be able to prevent deadlock

When do deadlocks happen?
Suppose
Process 1 holds resource A and resource B
Process 2 holds B and requests A

Both can be blocked, with neither able to proceed
Deadlocks occur when …
Processes are granted

What is a deadlock?

Formal definition:
"A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause."

Usually, the event is release of a currently held resource
In deadlock, none of the processes can
Run Release resources
Be awakened

Four conditions for deadlock

Mutual exclusion
Each resource is assigned to at most one process
Hold and wait
A process holding resources can request more resources
No preemption
Previously granted resources cannot be forcibly taken away
Circular wait
There must be a circular chain of 2 or more processes where each is waiting for a resource held by the next member of the chain

Resource allocation graphs

Resource allocation modeled by directed graphs
Example 1:
Resource R assigned to process A
Example 2:
Process B is requesting / waiting for resource S
Example 3:
Process C holds T, waiting for U
Process D holds U, waiting for T
C and D are in deadlock

Dealing with deadlock

How can the OS deal with deadlock?
Ignore the problem altogether!
Hopefully, it'll never happen …
Detect deadlock & recover from it
Dynamically avoid deadlock
Careful resource allocation

Prevent deadlock
Remove at least one of the four necessary conditions
We'll explore these tradeoffs

Getting into deadlock

A	B	C
Acquire R Acquire S Release R Release S	Acquire S Acquire T Release S Release T	Acquire T Acquire R Release T Release R

Lives of the Others

The Ostrich Algoruthm
Pretend there's no problem
Reasonable if
Deadlocks occur very rarely
Cost of prevention is high
UNIX and Windows take this approach
Resources (memor

Not getting into deadlock
Many situations may result in deadlock (but don't have to)
In previous example, A could release R before C requests R, resulting in no deadlock
Can we always get out of it this way?
Find ways to:
Detect deadlock and reverse it
Stop it from happening in the first place

Detecting deadlocks using graphs

Process holdings and requests in the table and in the graph
Graph contains a cycle => deadlock
Easy to pick out by looking at it (in this case)
Need to mechanically detect deadlocks

Deadlock detection algorithm

General idea: try to find cycles in the resource allocation graph
Algorithm:depth-first search at each code
Mark arcs as they're traversed
Build list of visited nodes
If node to be added is already on the list, a cycle exists!
Cycle => deadlock

Recovering from deadlock

->Recovery through preemption

- Take a resource from some other process
- Depends on nature of the resource and the process

->Recovery through rollback

- Checkpoint a process periodically
- Use saved state to restart the process if it's in deadlock
- May present a problem if the process affects lots of "external" things

->Recovery through killing processes

- Crudest but simplest way to break a deadlock: kill one of the processes in the deadlock cycle
- Other processes can get its resources
- Try to choose a process that can be rerun from the start
    - Pick one that hasn't run too far already

Preventing Deadlock

Deadlock can be completely prevented
Ensure that at least one of the conditions for deadlock never occurs
Mutual exclusion
Circular wait
Hold & wait

Eliminating mutual exclusion

Some devices (such as printer) can be spooled

- Only the printer daemon uses printer resource
- This eliminates deadlock for printer

Not all devices can be spooled
Principle:

- Avoid assigning resource when not absolutely necessary
- As few processes as possible actually claim the resource

Serialization

Attacking "hold and wait"

Require processes to request resources before starting
A process never has to wait for what it needs
This can present problems
A process may not know required resources at start of run
This also ties up resources other processes could be using
Processes will tend to be conservative and request resources they might need
Variation: a process must give up all resources before making a new request
Process is then granted all prior resources as well as the new ones
Problem: what if someone grabs the resources in the meantime -- how can the process save its state?

Attacking “no preemption”

This is not usually a viable option

Consider a process given the printer

- Halfway through its jobs, take away the printer
- Confusion ensues!

May work for some resources

- Forcibly take away memory pages, suspending the process
- Process may be able to resume with no ill effects

Attacking “circular wait”

Assign an order to resources

Always acquire resources in numberical order

Need not acquire them all at once

Circular wait is prevented A process holding resource n can’t wait for resource m if m<n

No way to complete a cycle!

- Place processes above the highest resource they hold and below any they’re requestingf

All arrows point up

What does FreeBSD do?

What resources are at issue?

Locks and semaphores: one holder at a time1

Physical resources: not typically

Deadlock prevention: summary

Condition	**Prevented by**
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	take resources away if there’s not a complete set
Circular wait	Order resources numerically

Example: two-phase locking

Phase One

Process tries to lock all data it needs, one at a time

If needed data found locked, start over (no real work done in phase one)

Phase Two

Banker’s algorithm for a single resource

Banker’s algorithm with multiple resources

Starvation

Algorithm to allocate a resource

Give the resource to the shortest job first

Works great for multiple short jobs in a system

May cause long jobs to be postponed indefinitely
Even though not blocked
Solution
First-come, first-serve policy
Starvation

Livelock

-> Sometimes, processes can still run, but not make progress
-> Example: two processes want to use resources A and B
    P0 gets A, P1 gets B
    Each realizes that a deadlock will occur if they proceed as planned
    P0 drops A, P1 drops B
    P0 gets B, P1 gets A
    Same problem as before
    This can go on for a very long time…
-> Real-world example: Ethernet transmission collisions
    If there's a "collision" on the wire, wait and try again
    Multiple processes waited the exact same amount of time …

Exam
Lecture first One hour long
Short test
Bring a page of notes -> not from the slides (create your own notes)
Double sided
8 pt font
Deadlocks
Scheduling, IPC, Processes
Little memory (easy)