Outline for today

- Objective:
  - Background on deadlock
  - Pulse
    - Speculative execution
    - Virtual Machines and Xen
- Administrative:
  - Make teams for programming projects

Background on Deadlock

Dealing with Deadlock

It can be ***prevented*** by breaking one of the prerequisite conditions (review):

- Mutually exclusive use of resources
  - Example: Allowing shared access to read-only files (readers/writers problem from readers point of view)
- circular waiting
  - Example: Define an ***ordering*** on resources and acquire them in order (lower numbered fork first)
- hold and wait
- no pre-emption

Dealing with Deadlock (cont.)

Let it happen, then ***detect*** it and ***recover***

- via externally-imposed preemption of resources

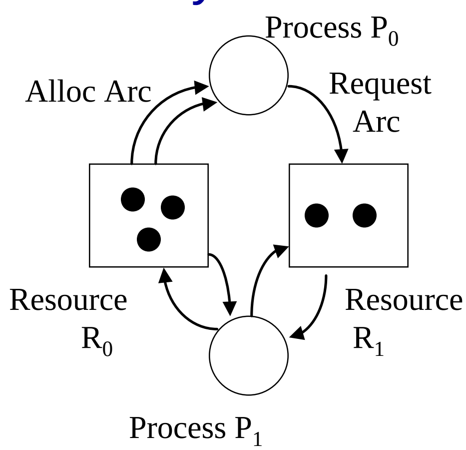
Avoid dynamically by monitoring resource requests and denying some.

- Banker's Algorithm **...**

Deadlock Theory

State of resource allocation captured in ***Resource Graph***

- Bipartite graph model with a set **P** of vertices representing processes and a set **R** for resources.
- Directed edges
  - R_i –> P_j means R_i alloc to P_j
  - P_j –> R_i means P_j requests R_i
- Resource vertices contain ***units*** of the resource



Reusable Resources

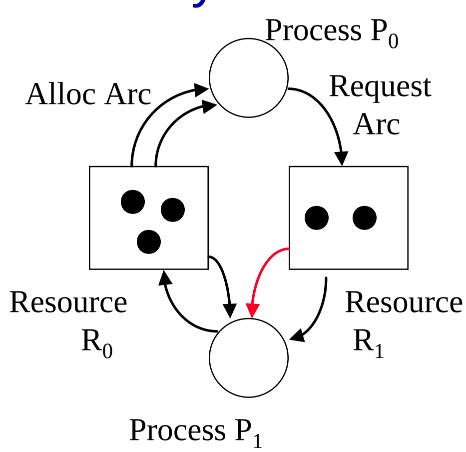
Deadlock Theory

State transitions by operations:

- Granting a request
- Making a new request if all outstanding requests satisfied

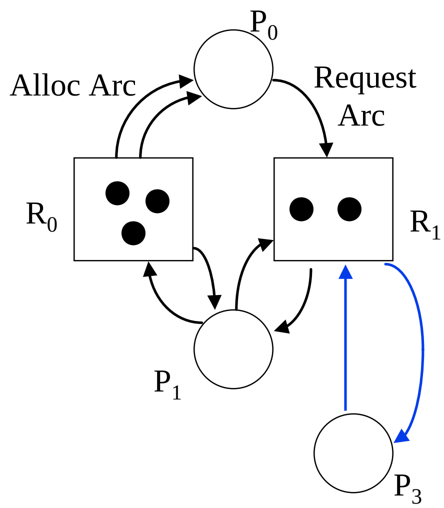
Deadlock defined on graph:

- P_i is ***blocked*** in state S if there is no operation P_i can perform
- P_i is ***deadlocked*** if it is blocked in all reachable states from S
- S is ***safe*** if *no* reachable state is a ***deadlock state*** (i.e., having some deadlocked



Deadlock Theory

- Cycle in graph is a necessary condition
  - no cycle –> no deadlock.
- No deadlock iff graph is ***completely reducible***
  - Intuition: Analyze graph, asking if deadlock is *inevitable* from this state by simulating most favorable state transitions.

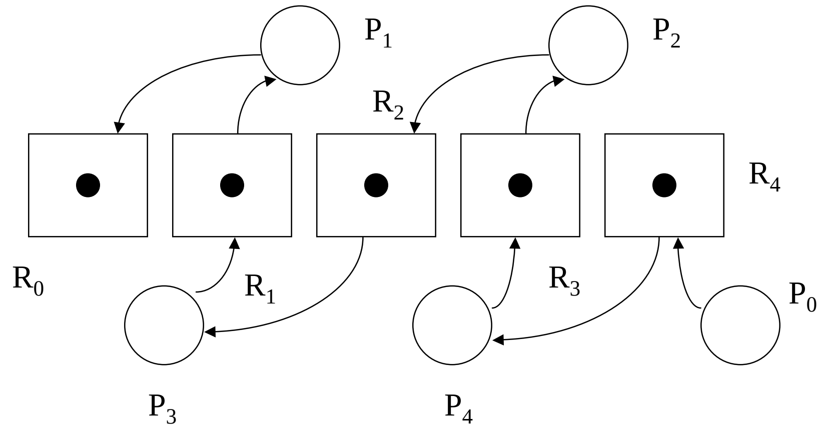


Deadlock Detection Algorithm

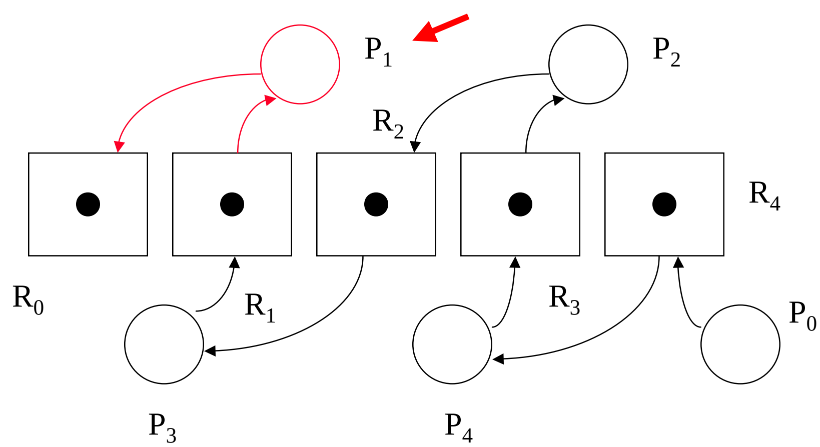
Let U be the set of processes that have yet to be reduced. Initially U = P. Consider only *reusable* resources.

while (there exist *unblocked* processes in U)
{ Remove unblocked P_i from U;
Cancel P_i's outstanding requests;
Release P_i's allocated resources;
/* possibly unblocking other P_k in U */}
if (U != λ) signal deadlock;

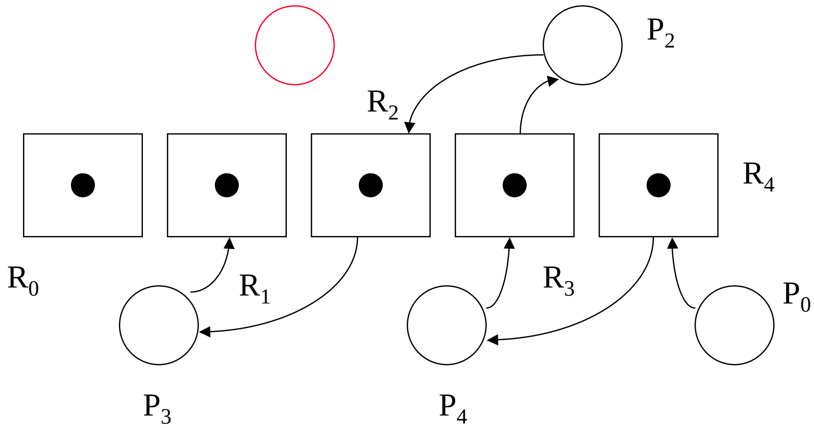
Deadlock Detection Example



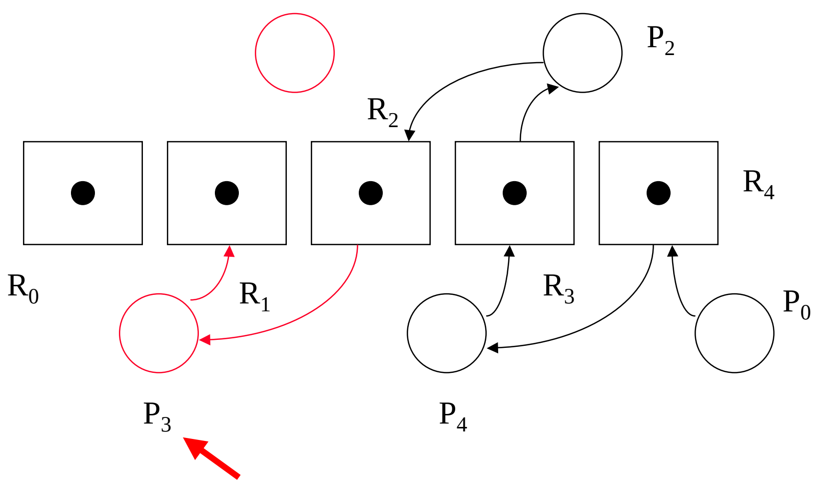
Deadlock Detection Example



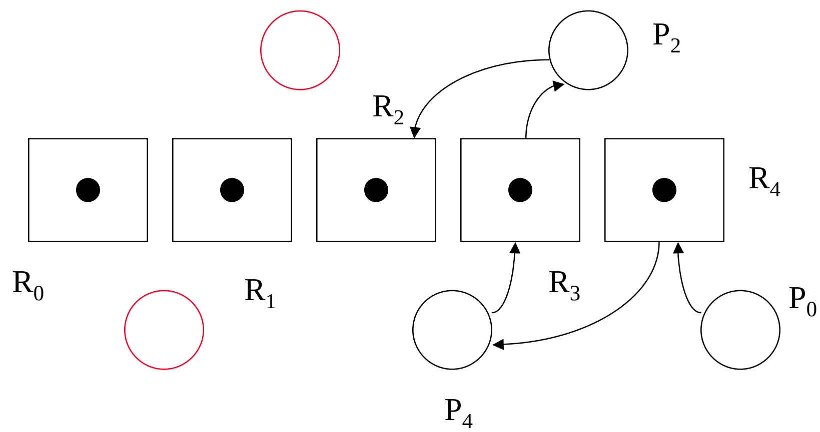
Deadlock Detection Example



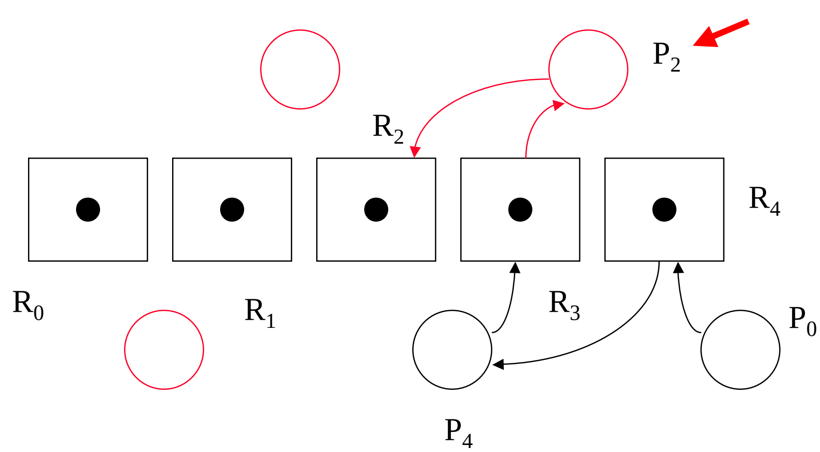
Deadlock Detection Example



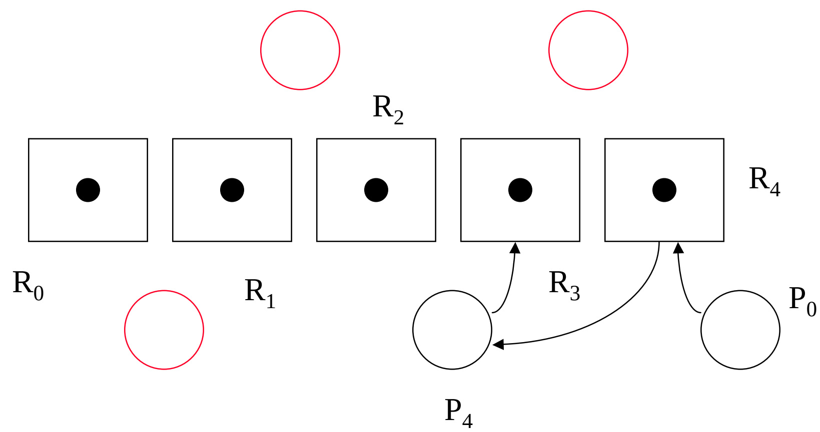
Deadlock Detection Example



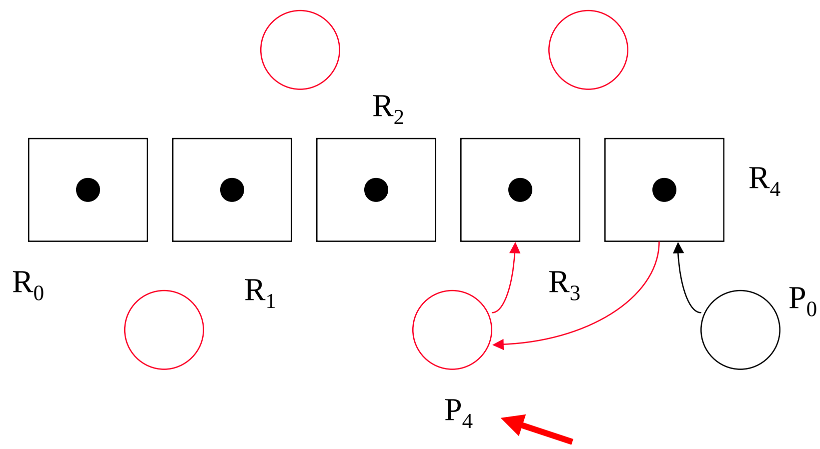
Deadlock Detection Example



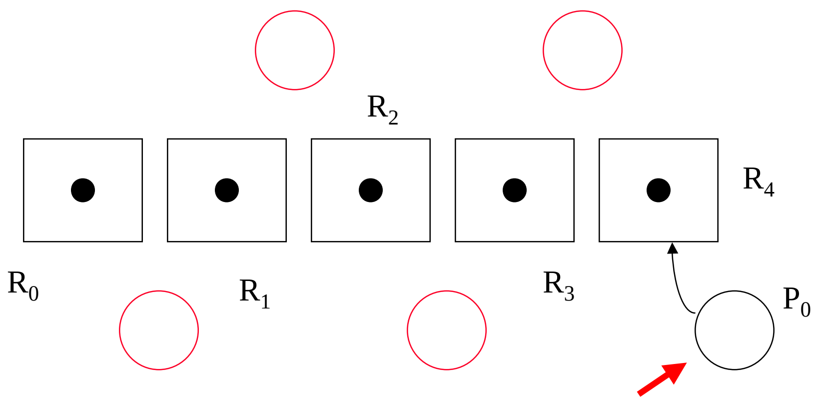
Deadlock Detection Example



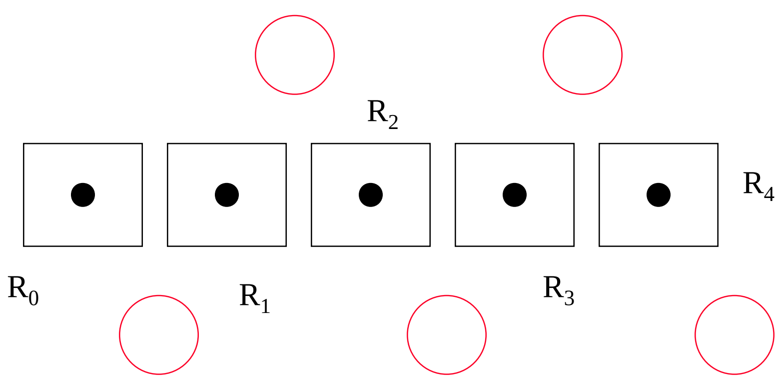
Deadlock Detection Example



Deadlock Detection Example

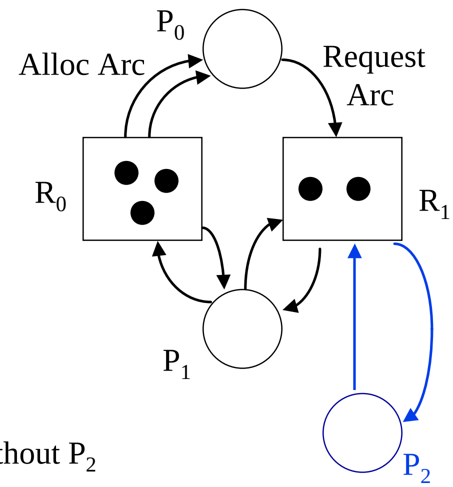


Deadlock Detection Example



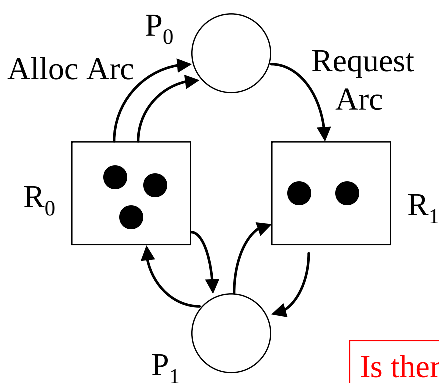
Completely Reducible

Another Example



With and without P_2

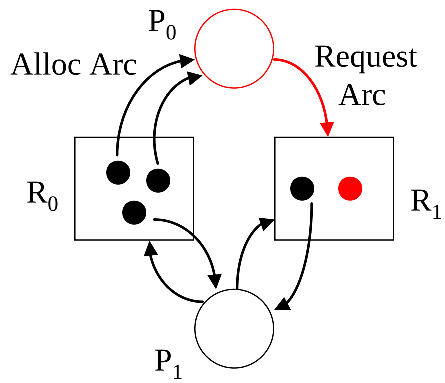
Another Example



Is there an unblocked process to start with?

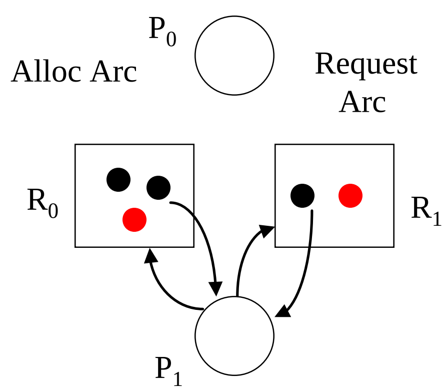
With and without P_2

Another Example



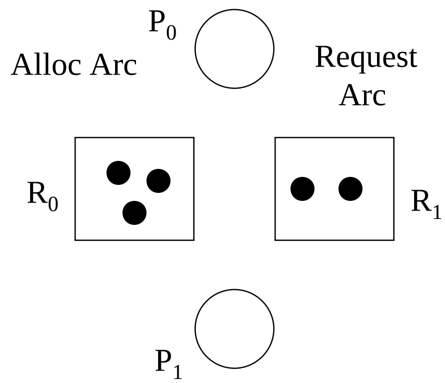
With and without P_2

Another Example



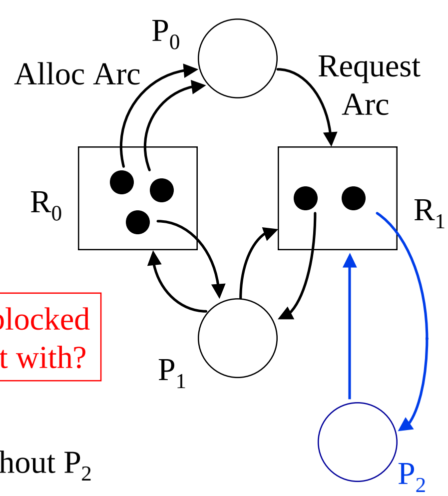
With and without P_2

Another Example



With and without P_2

Another Example

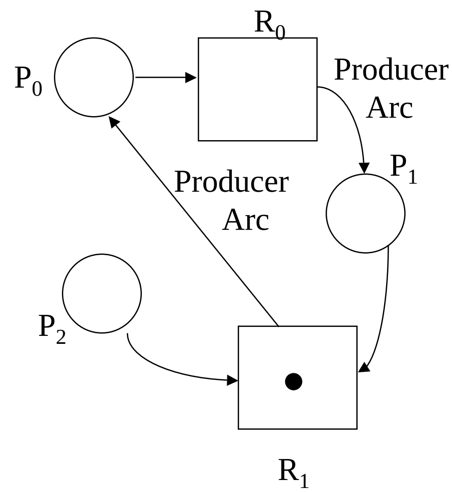


Is there an unblocked process to start with?

With and without P_2

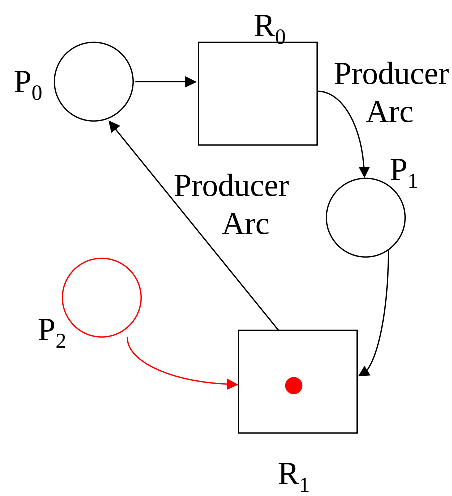
Consumable Resources

- Not a fixed number of units, operations of producing and consuming (e.g. messages)
- Ordering matters on applying reductions
  - Reducing by producer makes "enough" units, ω



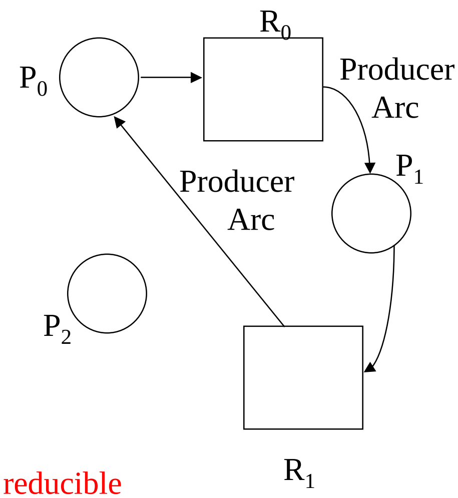
Consumable Resources

- Not a fixed number of units, operations of producing and consuming (e.g. messages)
- Ordering matters on applying reductions
  - Reducing by producer makes "enough" units, ω
  - Start with P_2



Consumable Resources

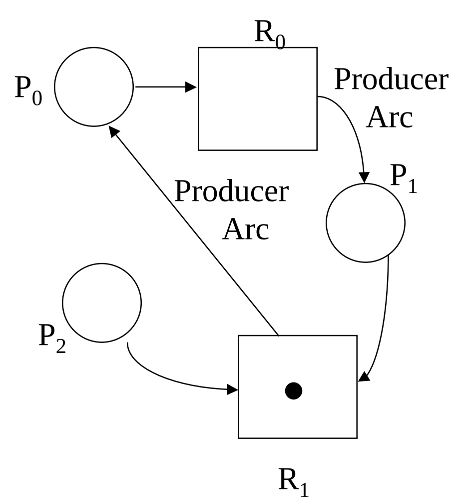
- Not a fixed number of units, operations of producing and consuming (e.g. messages)
- Ordering matters on applying reductions
  - Reducing by producer makes “enough” units, ω
  - Start with P_2



Not reducible

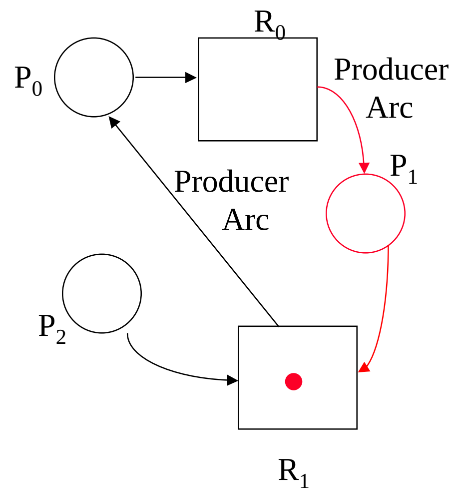
Consumable Resources

- Not a fixed number of units, operations of producing and consuming (e.g. messages)
- Ordering matters on applying reductions
  - Reducing by producer makes “enough” units, ω
  - Start with P_2



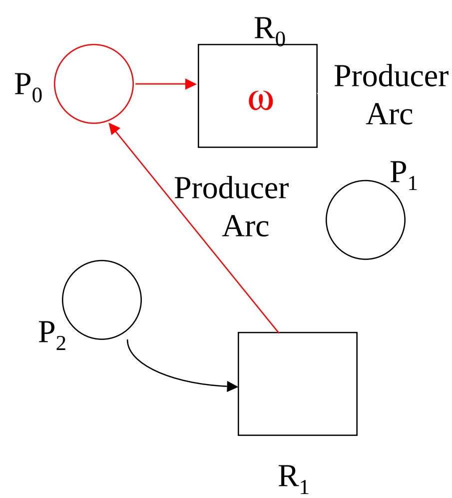
Consumable Resources

- Not a fixed number of units, operations of producing and consuming (e.g. messages)
- Ordering matters on applying reductions
  - Reducing by producer makes "enough" units, ω
  - Start with P_2
  - ~~Start with P_1~~



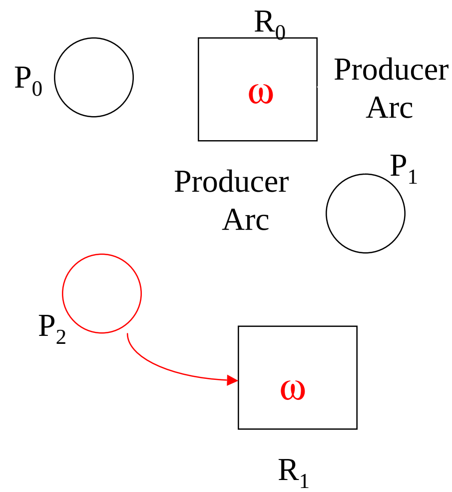
Consumable Resources

- Not a fixed number of units, operations of producing and consuming (e.g. messages)
- Ordering matters on applying reductions
  - Reducing by producer makes "enough" units, ω
  - Start with P_1



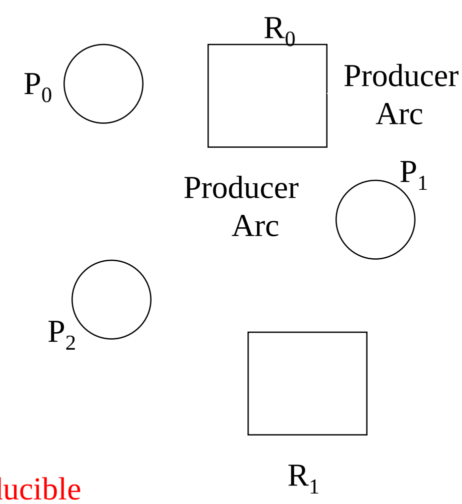
Consumable Resources

- Not a fixed number of units, operations of producing and consuming (e.g. messages)
- Ordering matters on applying reductions
  - Reducing by producer makes “enough” units, ω
  - Start with P_1



Consumable Resources

- Not a fixed number of units, operations of producing and consuming (e.g. messages)
- Ordering matters on applying reductions
  - Reducing by producer makes “enough” units, ω
  - Start with P_1



Reducible

Deadlock Detection & Recovery

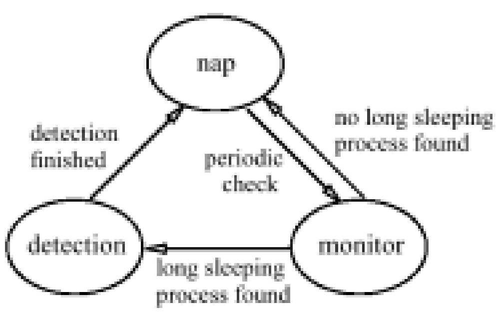
- Continuous monitoring and running this algorithm are expensive.
- What to do when a deadlock is detected?
  - Abort deadlocked processes (will result in restarts).
  - Preempt resources from selected processes, rolling back the victims to a previous state (undoing effects of work that has been done)
  - Watch out for starvation.

Pulse

Goal

- To increase the kinds of deadlocks that can be detected dynamically
- Uses high-level speculative execution to go forward to discover dependencies

Overview of Pulse

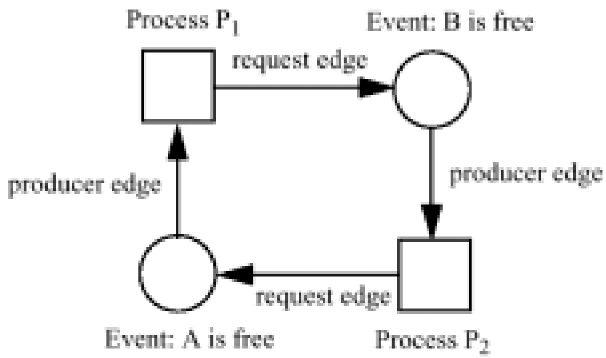


- Kernel daemon process

  Presence of long-sleeping processes trigger detection

  Detection mode
  - Identify processes and events awaited
  - Fork speculative processes to see what events they generate in the future

Creating General Resource Graph with Consumable Resources



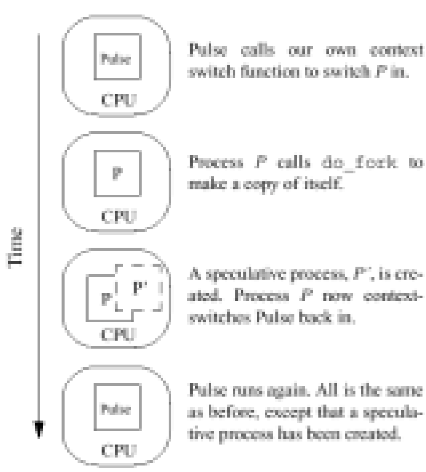
Details of Graph Construction

- Process and Event nodes
  - Those processes blocked a long time.
  - Events – all blocking system calls modified to record the events for which caller waits (resource, condition <op, val>)
- Edges
  - Request edges generated with event nodes.
  - Producer edges result from speculation
    - Recorded in event buffer until speculative processes terminate (normally, full buffer, timeout)
    - Modifying all system calls that unblock the blocking ones
- Cycle detection on finished graph

Safe Speculation

- Must not modify state of any other process
  - Fork with copy-on-write enabled
  - Can not change shared kernel data structures
  - Can not write to files
  - Can not send signals to another process
- Pretend properly that we get unblocked ourselves
  - Not really reading input data if that's what we were waiting for (so data dependent branches won't be "right")
  - Must pretend that conditions true (in case of while loop in application code)

Tricks of Forking Blocked Processes



New process is forced to run ret_from_spec_fork

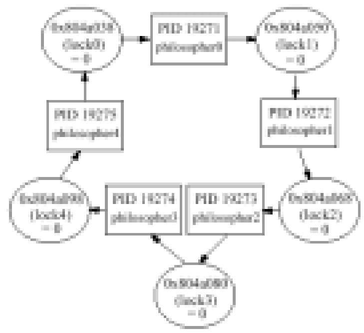
Fake the awaited event

syscall_exit with success

5 Dining Philosophers

```
while (1) {
    think();
    lock(fork[i]);                  // take left fork
    lock(fork[(i + 1) % 5];         // take right fork
    eat();
    unlock(lock[i]);                // put left fork
    unlock(lock[(i + 1) % 5]);      // put right fork
}
```

Figure 6: The code of philosopher i.

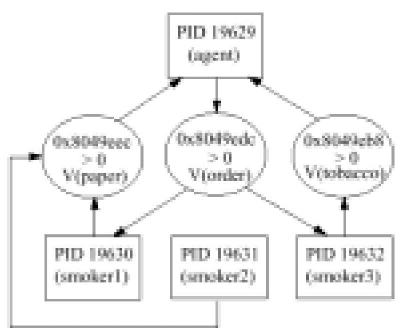


Smoker’s Problem

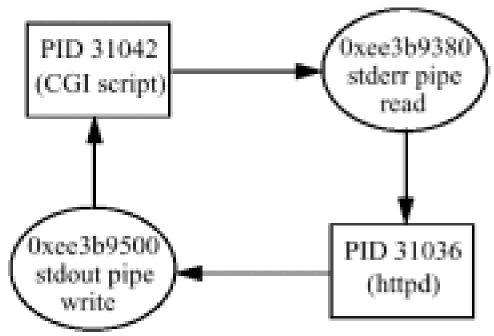
Suppose agent releases tobacco and matches

```
smoker 1              smoker 2              smoker 3
while (1) {           while (1) {           while (1) {
  P(tobacco)            P(paper) // block   P(matches)
  P(paper) // block     P(matches)            P(tobacco) // block
  V(order)              V(order)              V(order)
}                     }                     }

        agent
        while (1) {
          P(order) // block
          V(one of tobacco, paper, order at random)
          V(one of the three at random but not above)
        }
```



Apache Bug



Limitations

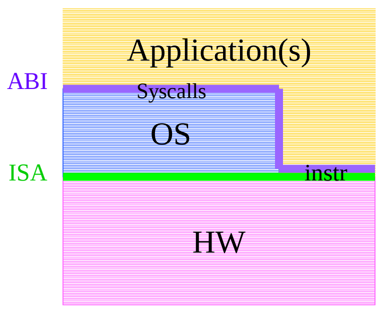
- False positives
  – Since everything appears as consumable resources, Pulse could find more than one producer edge (and extra cycles)
  – Since more than single unit resources – a cycle is really just necessary not sufficient
- False negatives
  – Self-breaking mechanisms
  – Events that never occur (no unlocks)

Extensions

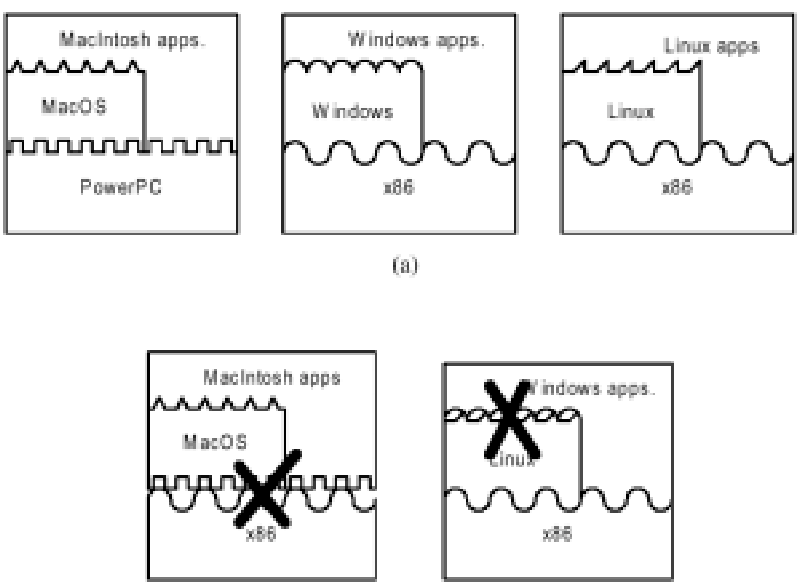
- Spinning synchronization – we just need to identify spinning as form of blocking by the system – instrument calls
- Kernel deadlocks – use virtual machine to speculatively execute a kernel instance.

Intro to Virtual Machines

Traditional Multiprogrammed OS



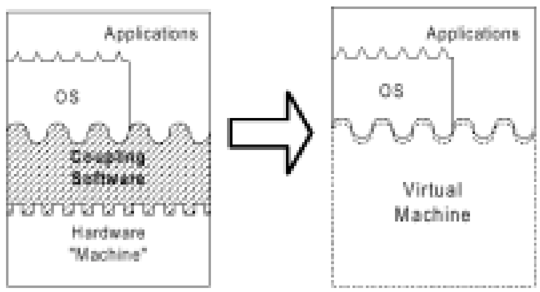
- Multiple applications running with the abstraction of dedicated machine provided by OS
- Pass through of non-privileged instructions
- ISA – instruction set architecture
- ABI – application binary interface



(a)

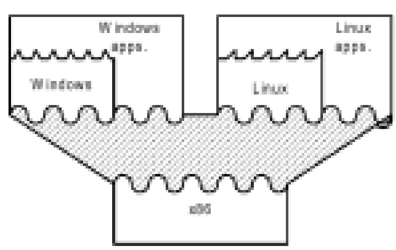
(b)

Virtualization Layer

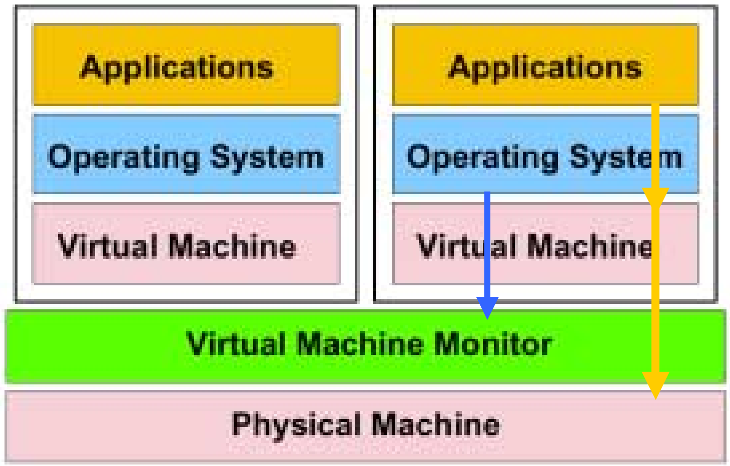


Virtual Machines

- History: invented by IBM in 1960's
- Fully protected and isolated copy of the physical machine providing the abstraction of a dedicated machine
- Layer: Virtual Machine Monitor (VMM)
- Replicating machine for multiple OSs
- Security Isolation



Virtual Machine Monitor



Issues

- Hardware must be fully virtualizable – all sensitive (privileged) instructions must trap to VMM
  - X86 is not fully virtualizable
- In traditional model, all devices need drivers in VMM
  - PCs have lots of possible devices – leverage the host OS for its drivers => hosted model

Xen

Paravirtualization

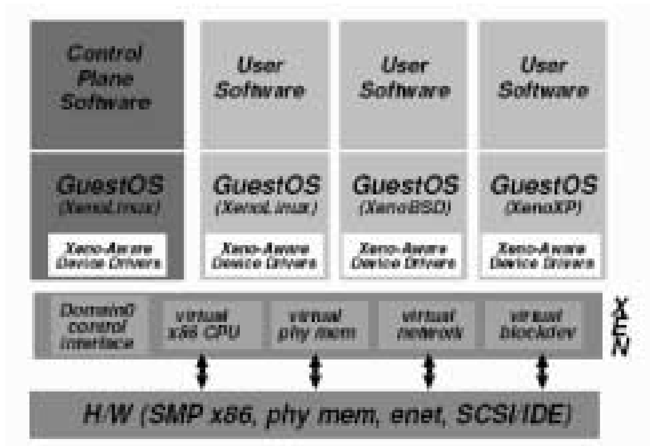
- A virtual machine that is not identical to real hardware
- Does not require changes to application interface (support unmodified user code).
- Does require source modifications to kernel – XenoLinux.

Structure

Privilege ring 3

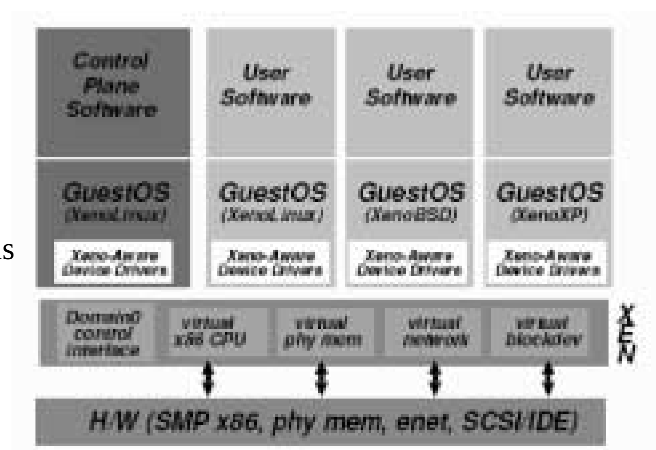
Privilege ring 1

Privilege ring 0



Structure

hypercalls ↓



↑ events