

Chapter 7: Deadlock

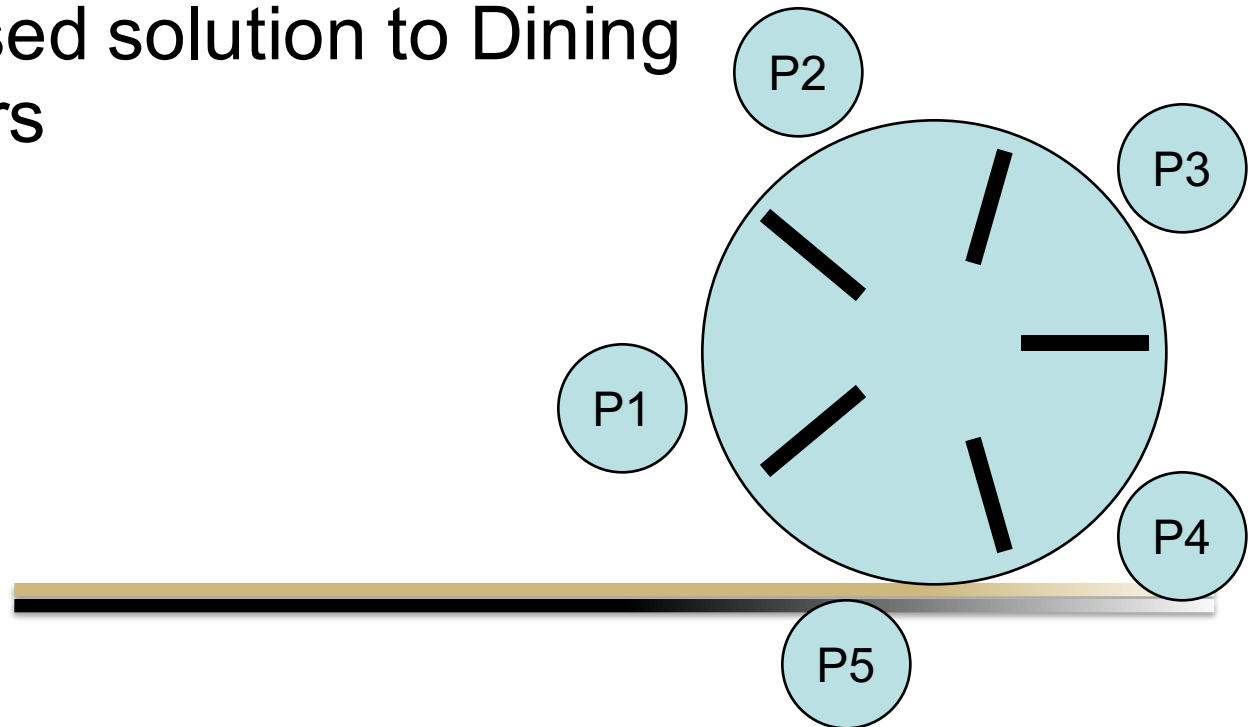
CSCI 3753 Operating Systems

William Mortl, MS and Rick Han, PhD



Recap

- Monitors for implicit mutual exclusion
- Condition variables for ordering
 - `x.wait()`
 - `x.signal()` differs from semaphore's `signal()`
- Monitor-based solution to Dining Philosophers



Complete Monitor-based Solution to Dining Philosophers

```
monitor DP {
    status state[5];
    condition self[5];

    Pickup(int i) {
        state[i] = hungry;
        test(i);
        if (state[i] != eating)
            self[i].wait;
    }

    test(int i) {
        if (state[(i+1)%5] != eating &&
            state[(i-1)%5] != eating &&
            state[i] == hungry) {

            state[i] = eating;
            self[i].signal();
        }
    }

    Putdown(int i) {
        state[i] = thinking;
        test((i+1)%5);
        test((i-1)%5);
    }

    init() {
        for i = 0 to 4
            state[i] = thinking;
    }

} // end of monitor
```

- Pickup(), Putdown() and test() are all mutually exclusive, i.e. only one at a time can be executing
- Verify that this monitor-based solution is
 - deadlock-free
 - mutually exclusive in that no 2 neighbors can eat simultaneously



Deadlock Can Easily Occur

- Carefully engineered synchronization solutions to avoid deadlock
 - The 3 classic synchronization problems like Dining Philosophers, Readers/Writers, and Bounded Buffer P/C
- saw earlier that semaphores provide mutual exclusion, but can introduce deadlock
 - 2 tasks, each desires a resource locked by the other process
 - Circular dependency
 - can occur easily due to programming errors, e.g. by switching order of P and V, etc.



Deadlock Is Hard To Anticipate

- Difficult to anticipate by looking at code of a single process or thread
 - deadlock is a higher-level concept that involves the distributed behavior of multiple processes/threads, e.g. a lengthy circular dependency across many Dining Philosophers
- Example:
 - a Web server talks with a MySQL database which talks with a data mining process which communicates back to the Web server
 - Each task is written by a different programmer with no advanced knowledge how it might be used with other tasks
- deadlock is difficult to anticipate, detect, reproduce, prevent, avoid, and recover from



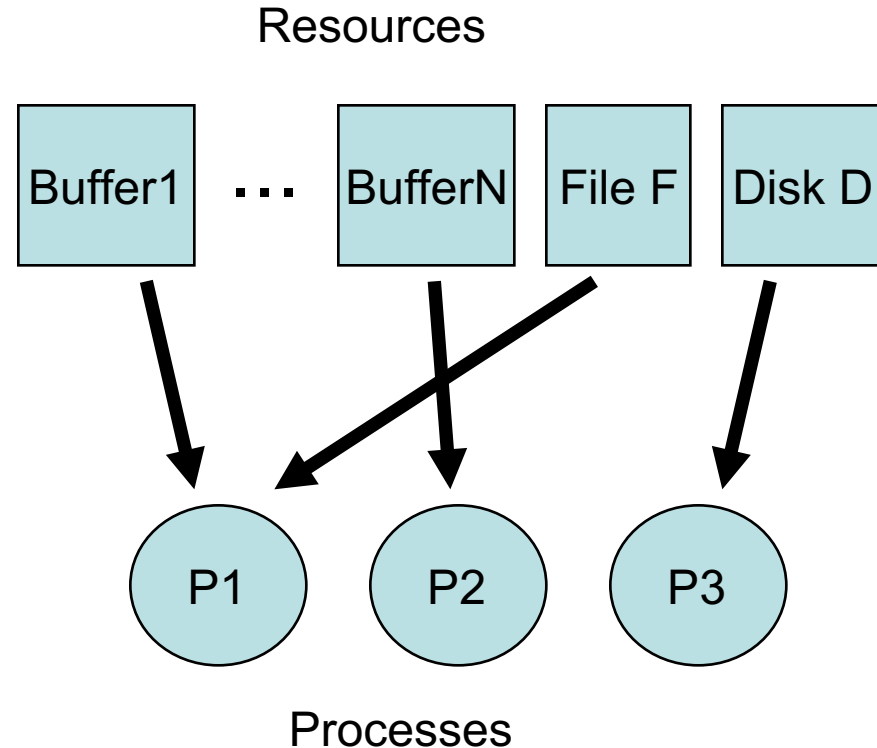
Deadlock: General Solution?

- Want a general solution to deadlock that is not restricted to the solutions for the 3 classic problems of DP, R/W, and BB P/C
- A set of processes is in a deadlock state when every process in the set is waiting for an event (e.g. release of a resource) that can only be caused by another process in the set
 - You have a circular dependency
- multithreaded and multi-process applications are good candidates for deadlock
 - thread-thread deadlock within a process
 - process-process deadlock



Modeling Deadlock

- Develop a model so we can see circular dependency
 - to use a resource, a process must
 1. request() a resource -- must *wait* until it's available
 2. use() or hold() a resource
 3. release() a resource
 - thus, we have resources and processes
 - Most of the following discussion will focus on reusable resources



P1 holds Buffer 1 and File F
P2 holds Buffer N
P3 holds Disk D



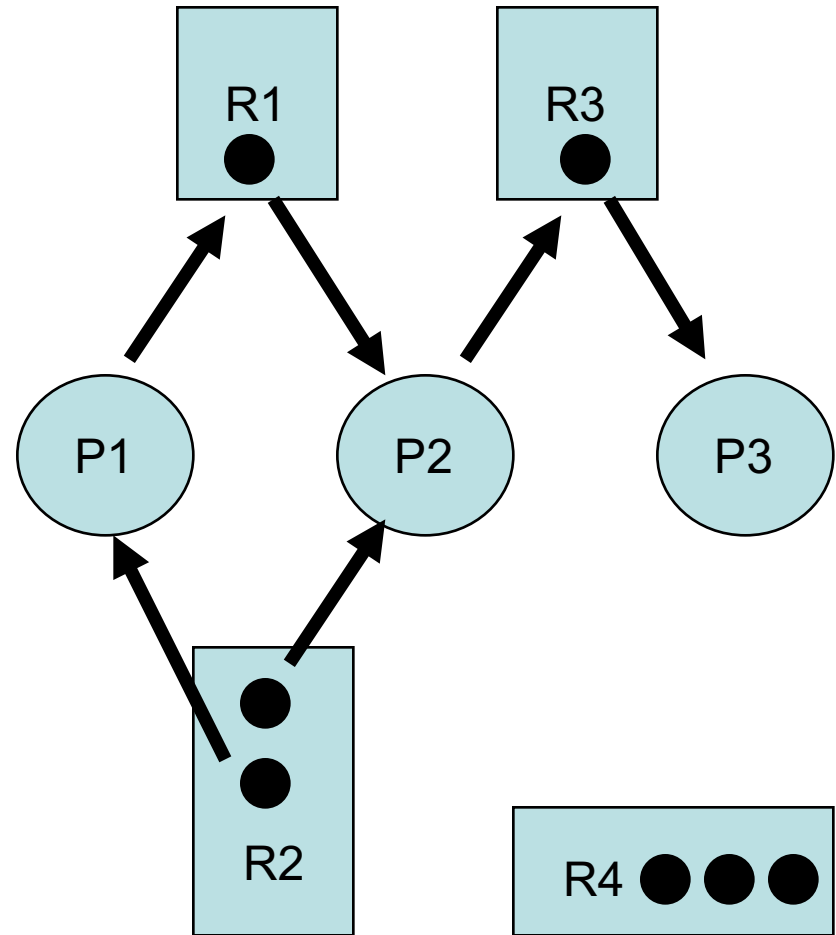
Modeling Deadlock

- a *resource allocation graph* can be used to model deadlock
 - try to represent deadlock by a *directed graph* $D(V,E)$, consisting of
 - vertices V : namely processes and resources
 - and edges E :
 - a request() for a resource R_j by a process P_i is signified by a directed arrow from process $P_i \rightarrow R_j$
 - a process P_i will hold() a resource R_j via a directed arrow $R_j \rightarrow P_i$



Modeling Deadlock

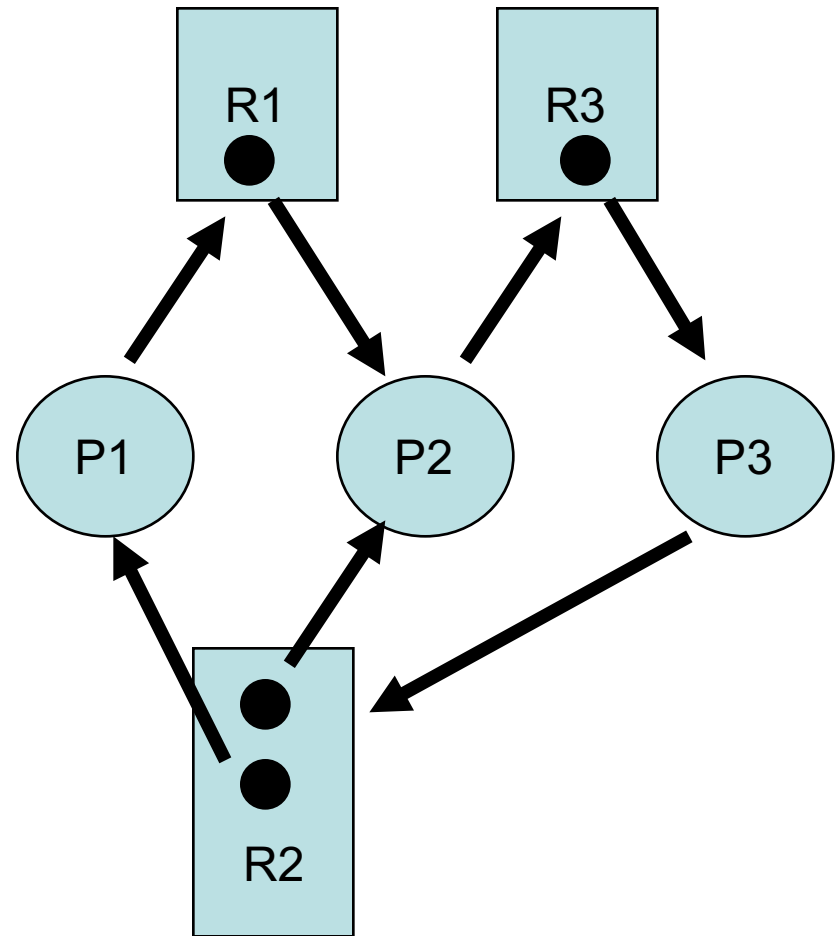
- Example 1:
 - P1 wants resource R1 but that is held by P2
 - P2 wants resource R3 but that is held by P3
 - Also, P1 holds an *instance* of resource R2, and
 - P2 holds an instance of R2
 - There is no deadlock
 - if the graph contains no cycles or loops, then there is no deadlock



Modeling Deadlock

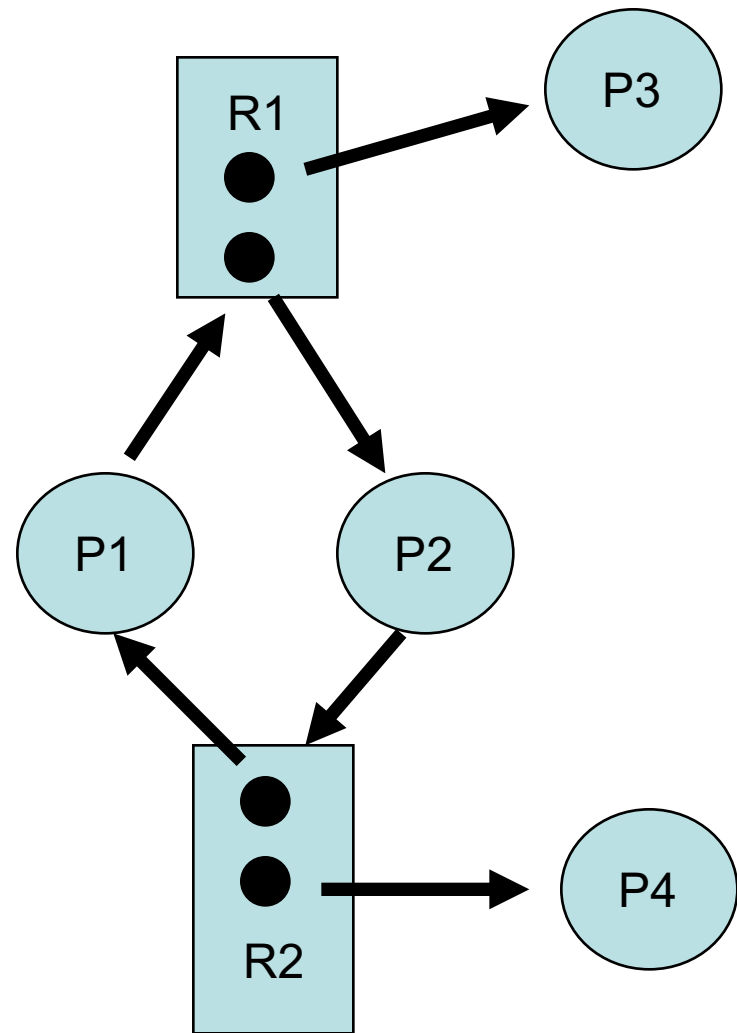
- Example 2:

- same graph as before, except now P3 requests instance of R2
- Deadlock occurs!
 - P3 requests R2, which is held by P2, which requests R3, which is held by P3 - this is a loop
 - $P_3 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3$
 - If P1 could somehow release an instance of R2, then we could break the deadlock
 - But P1 is part of a second loop:
 - $P_3 \rightarrow R_2 \rightarrow P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3$
 - So P1 can't release its instance of R2
- if the graph contains cycles or loops, then there *may be the possibility* of deadlock
 - but does a loop guarantee that there is deadlock?



Modeling Deadlock

- Example 3:
 - there is a loop:
 - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_1$
 - In this case, there is no deadlock
 - either P3 can release an instance of R1, or P4 can release an instance of R2
 - this breaks any possible deadlock cycle
 - if the graph contains cycles or loops, then there *may be the possibility* of deadlock, but this is not a guarantee of deadlock

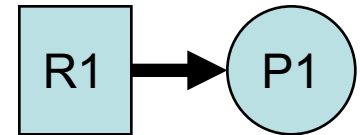


Necessary Conditions for Deadlock

- The following 4 conditions must hold simultaneously for deadlock to arise:

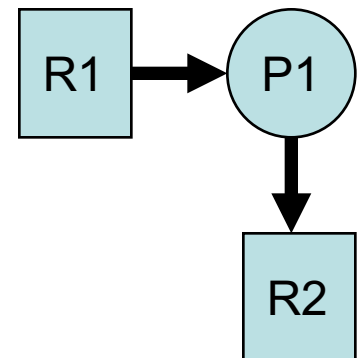
1. Mutual exclusion

- at least 1 resource is held in a non-sharable mode. Other requesting processes must wait until the resource is released



2. Hold and wait

- a process holds a resource while requesting (and waiting for) another one

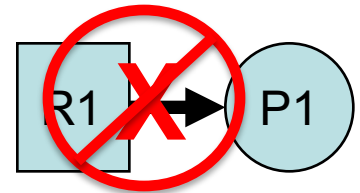


Necessary Conditions for Deadlock

- The following 4 conditions must hold simultaneously for deadlock to arise: (continued)

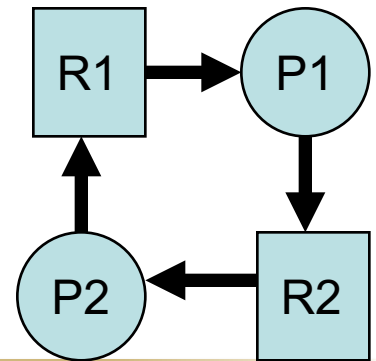
3. No preemption:

- resources cannot be preempted and can only be released voluntarily by the process holding them, after the process is finished. No OS intervention is allowed. A process cannot withdraw its request.



4. Circular wait

- A set of n waiting processes $\{P_0, \dots, P_{n-1}\}$ must exist such that P_i waits for a resource held by $P_{(i+1)\%n}$.



Solutions to Handling Deadlocks

1. Prevention by OS

- provide methods to guarantee that at least 1 of the 4 necessary conditions for deadlock does not hold

2. Avoidance by OS

- the OS is given advanced information about process requests for various resources
- this is used to determine whether there is a way for the OS to satisfy the resource requests and avoid deadlock



Solutions to Handling Deadlocks (2)

3. Detection and Recovery by OS

- Analyze existing system resource allocation, and see if there is a sequence of releases that satisfies every process' needs.
- If not, then deadlock is detected, so must recover – drastic action needed, like killing the affected processes!



Solutions to Handling Deadlocks

(3)

4. Application-level solutions (OS Ignores and Pretends)

- the most common approach, e.g. UNIX and Windows, based on the assumption that deadlock is relatively infrequent
- it's up to the application programmer to implement mechanisms that prevent, avoid, detect and deal with application-level deadlock
- Map your problem to known deadlock-free solutions: e.g. Bounded Buffer P/C, Readers/Writers problems, Dining Philosophers, ...



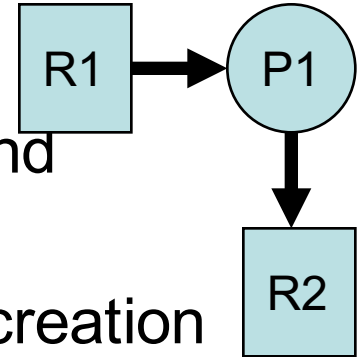
Deadlock Prevention: Mutual Exclusion

- Prevent the *mutual exclusion* condition #1 from coming true
 - This is opposite of our original goal, which was to provide mutual exclusion.
 - Also, many resources are non-sharable and must be accessed in a mutually exclusive way
 - example: a printer should print a file X to completion before printing a file Y. a printer should not print half of file X, and then print the first half of file Y on the same paper
 - thus, it is unrealistic to prevent mutual exclusion



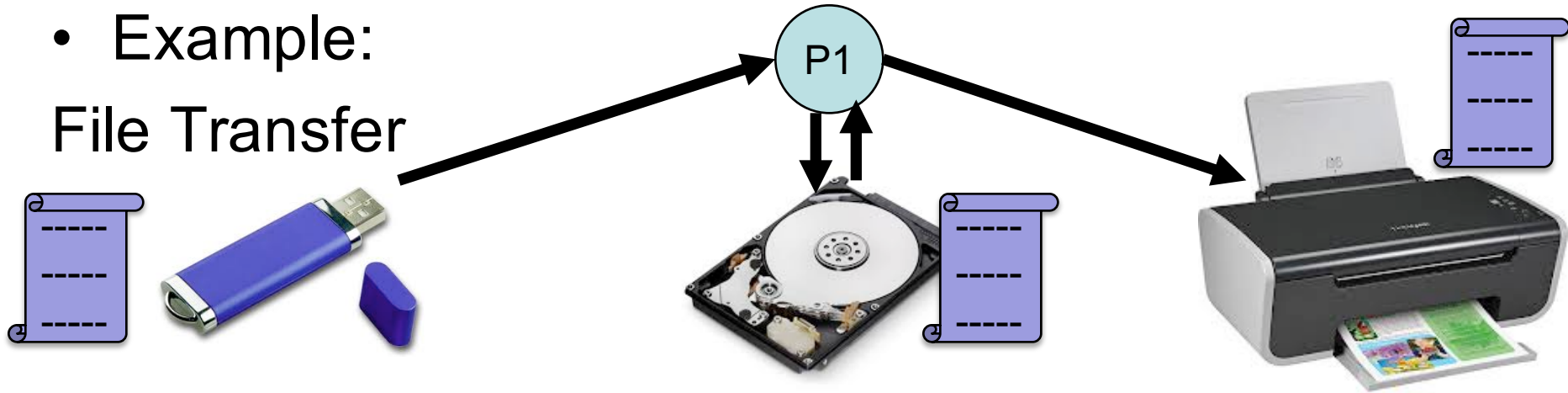
Deadlock Prevention: Hold and Wait

- Prevent the hold and wait condition #2 from coming true
 - prevent a process from holding resources and requesting others
 - Solution I: request all resources at process creation
 - Solution II: release all held resources before requesting a set of new ones simultaneously
 - Solution III: only allow a process to hold one resource at a time



Deadlock Prevention: Hold and Wait

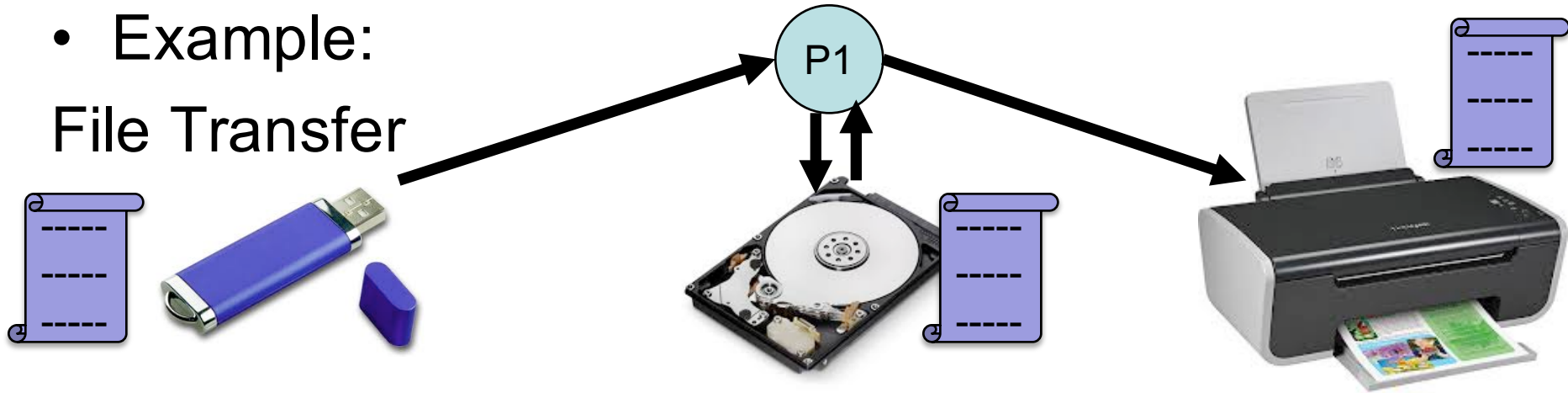
- Example:
File Transfer



- a process reads file from USB drive and writes it to hard drive, retrieves the file, then sends the file to the printer
 - Solution I: request the USB drive, hard drive, and printer at process creation

Deadlock Prevention: Hold and Wait

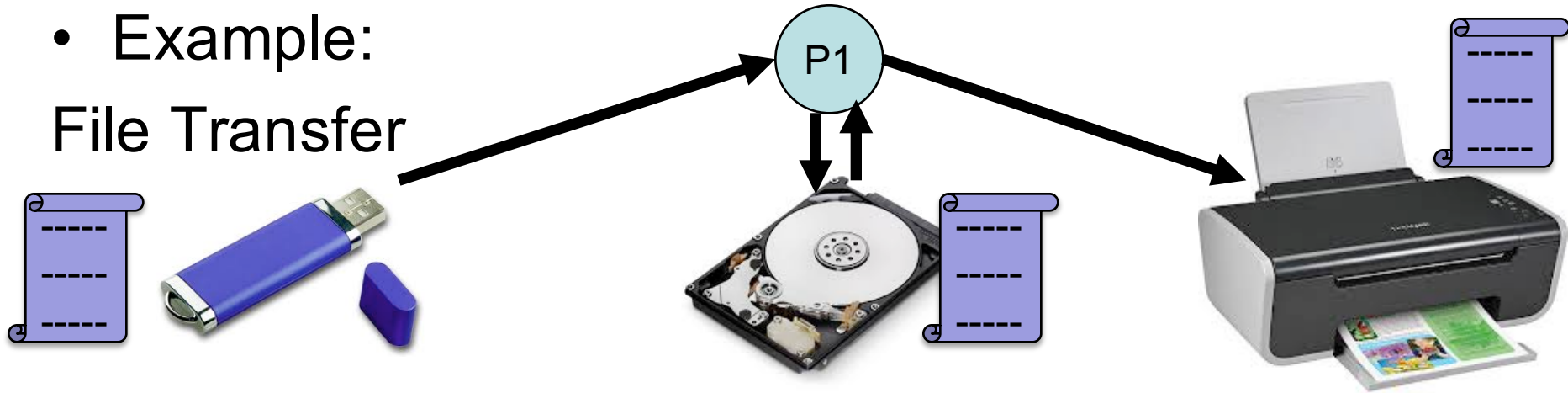
- Example:
File Transfer



- Solution II: divide task into self-contained stages that release all & then request all resources
 - obtain the USB and hard drive together for the file transfer, then release both together
 - next obtain the hard drive and printer together for the printing operation, then release both together

Deadlock Prevention: Hold and Wait

- Example:
File Transfer



– Solution III:

- Request the USB drive then release
- Request the hard drive then release
- Request the hard drive again then release
- Request the printer then release



Deadlock Prevention: Hold and Wait

- Disadvantages of Hold-and-wait solutions
 - Solution I: don't know in advance all resources needed
 - Solutions I & II: poor resource utilization
 - a process that is holding multiple resources for a long time may only need each resource for a short time during execution
 - Solution II: possible starvation
 - a process that needs several popular resources simultaneously may have to wait a very long time



Deadlock Prevention: Hold and Wait

- Disadvantages of Hold-and-wait solutions
 - Solution III: Some processing may require holding more than one resource at a time
 - e.g. writing a file to a printer may require locking both the file and the printer
 - Reading a file from a drive may require locking both the file and the drive



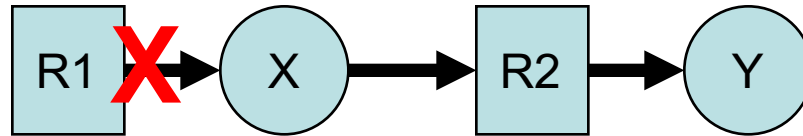
Deadlock Prevention: Hold and Wait

- Example: Dining Philosophers Problem prevented hold-and-wait – How?
 - Enforced a rule that either a philosopher picked up both chopsticks or none at all, i.e. all-or-nothing
 - Hence no holding one chopstick while waiting on the other chopstick



Deadlock Prevention: No Preemption

- Prevent the “No Preemption” condition #3 from coming true
 - allow resources to be preempted

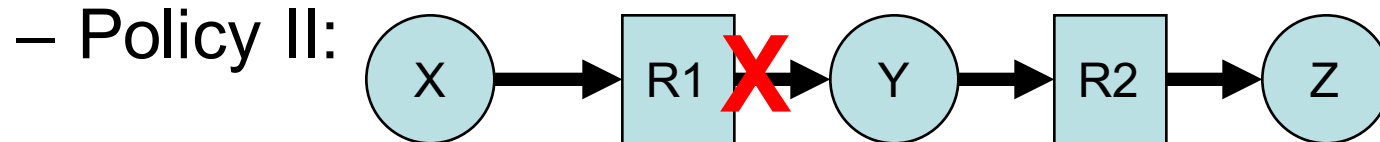


- Policy I:

- If a Process X requests a held resource, then all resources currently held by X are released.
- X is restarted only when it can regain all needed resources



Deadlock Prevention: No Preemption



- If a process X requests a resource held by process Y, then preempt the resource from process Y, but only if Y is waiting on another resource
- Otherwise, X must wait.
- the idea is if Y is holding some resources but is waiting on another resource, then Y has no need to keep holding its resources since Y is suspended



Deadlock Prevention: No Preemption

- Disadvantages:
 - these policies don't apply to all resources, e.g. printers should not be preempted while in the middle of printing, disks should not be preempted while in the middle of writing a block of data
 - can result in unexpected behavior of processes, since an application developer may not know a priori which policy is being used



Deadlock Prevention: Circular Wait

- Prevent the *circular wait* condition #4 from coming true
 - Solution I: a process can only hold 1 resource at a time
 - disadvantage: in some cases, a process needs to hold multiple resources to accomplish a task
 - Solution II: impose a total ordering of all resource types and require each process to request resources in increasing order
 - this prevents a circular wait - see next slide



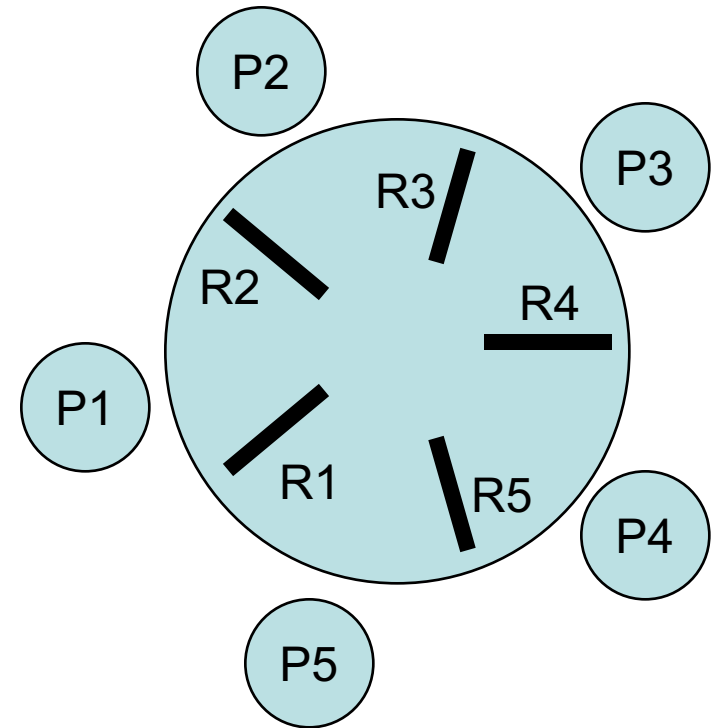
Deadlock Prevention: Circular Wait

- Solution II example:
 - Order all resources into a list: R_1, R_2, \dots, R_m , where $R_1 < R_2 < \dots < R_m$
 - tape drive = R_1 , disk drive = R_2 , printer = R_{10} , temporary buffer = R_{22}
 - Impose the rule that a process holding R_i can only request R_j if $R_j > R_i$
 - If a process P holds some R_k and requests R_j such that $R_j < R_k$, then the process must release all such R_k , acquire R_j , then reacquire R_k



Deadlock Prevention: Circular Wait

- Applying ordering of resources to break circular waiting in the Dining Philosophers Problem
 - $R1 < R2 < R3 < R4 < R5$
 - Deadlock happened when all processes first requested their right chopsticks, then requested their left chopsticks
 - Here, P1 to P4 can all request their right then left chopsticks
 - *But Process P5 requests its left (R1) then right (R5) chopstick due to ordering*
 - thus, P5 blocks on R1, not R5, which breaks any possibility of a circular deadlock - why?



Deadlock Prevention: Circular Wait

- Disadvantages of ordering resources:
 - can lead to poor performance, due to releasing and then reacquiring resources
 - Difficult to implement in a dynamic resource environment
 - Coming up with a global scheme for numbering resources



Deadlock Avoidance

- Goal: analyze the system state to see if there is a way to avoid deadlock.
- At startup, each process provides OS with information about all of its requests and releases for resources R_i
 - e.g. batch jobs know a priori which resources they'll request, when, and in which order
- OS decides whether deadlock will occur at run time



Deadlock Avoidance

- Disadvantage: need a priori info
- Simple strategy:
 - each process specifies a maximum claim
 - knowing all individual future requests and releases is difficult
 - Having each process estimate its maximum demand for resources is easier and not completely unreasonable
- A *resource allocation state* is defined by
 - # of available resources
 - # of allocated resources to each process
 - maximum demands by each process



Deadlock Avoidance

- A system is in a *safe* state if there exists a *safe sequence* of processes $\langle P_1, \dots, P_n \rangle$ for the current resource allocation state
 - A sequence of processes is safe if for each P_i in the sequence, the resource requests that P_i can still make can be satisfied by:
 - currently available resources + all resources held by all previous processes in the sequence $P_j, j < i$
 - If resources needed by P_i are not available, P_i waits for all P_j to release their resources



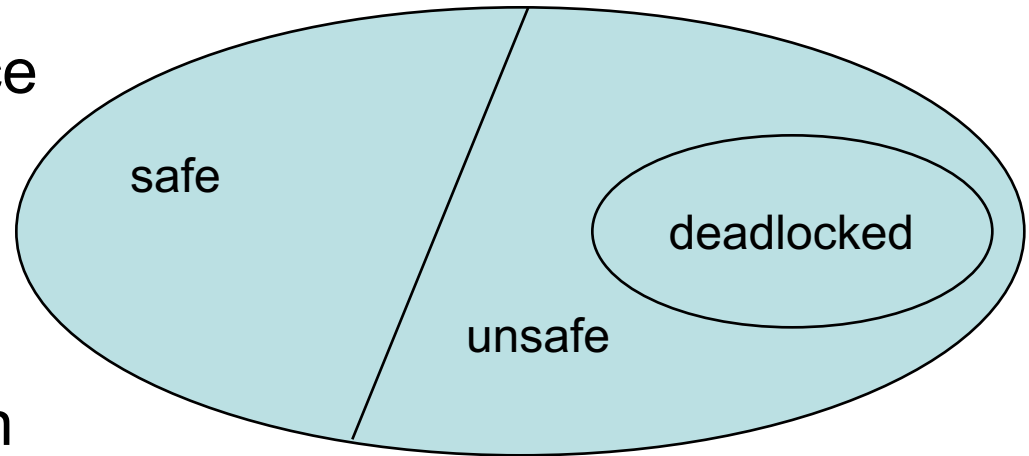
Deadlock Avoidance

- Intuition for a safe state: given that the system is in a certain state, we want to find at least one “way out of trouble”
 - i.e. find a sequence of processes that, even when they demand their maximum resources, won't deadlock the system
 - this is a worst-case analysis
 - it may be that during the normal execution of processes, none ever demands its maximum in a way that causes deadlock
 - to perform a more optimal (less than worst-case) analysis is more complex, and also requires a record of future accesses



Deadlock Avoidance

- A safe state provides a safe “escape” sequence
- A deadlocked state is unsafe
- An unsafe state is not necessarily deadlocked
- A system may transition from a safe to an unsafe state if a request for resources is granted
 - ideally, check with each request for resources whether the system is still safe



Deadlock Avoidance

- Example 1:
 - 12 instances of a resource
 - At time t_0 , P0 holds 5, P1 holds 2, P2 holds 2
 - Available = 3 free instances

processes	max needs	allocated
P0	10	5
P1	4	2
P2	9	2



Deadlock Avoidance

- Example 1 (cont):
 - Is the system in a safe state? Can I find a safe sequence?
 - Yes, I claim the sequence $\langle P1, P0, P2 \rangle$ is safe.
 - P1 requests its maximum (currently has 2, so needs 2 more) and holds 4, then there is only 1 free resource
 - Then P1 releases all of its resources, so 5 free
 - Next, P0 requests its max (currently has 5, so needs 5 more) and holds 10, so that now 0 free
 - Then P0 releases all its held resources, so 10 free
 - Next P2 requests its max of 9, leaving 3 free and then releases them all



Deadlock Avoidance

- Example 1 (cont):
 - Is the system in a safe state? Can I find a safe sequence?
 - Yes the sequence $\langle P1, P0, P2 \rangle$ is safe, and is able in the worst-case to request maximum resources for each process in the sequence, and release all such resources for the next process in the sequence
 - Can this system avoid deadlock? Yes, we can find a safe sequence.

