

---

# CSC 369H1S

Week 4: Monitors, Transactions &  
Deadlocks

Reading: Text, 2.3-2.4, Ch. 3

# Plan for Today

- Final synchronization strategies
  - Monitors
  - Transactions
- Dealing with deadlock
  - Defining deadlock
  - Conditions for deadlock to occur
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection (and recovery)

# But First...

- Assignment 1
  - Now available. Due Friday, Feb. 16
  - If you asked for help finding a group, please see me after class.
- We will go over the code reading questions and lock/cv implementations in tutorial next Wednesday

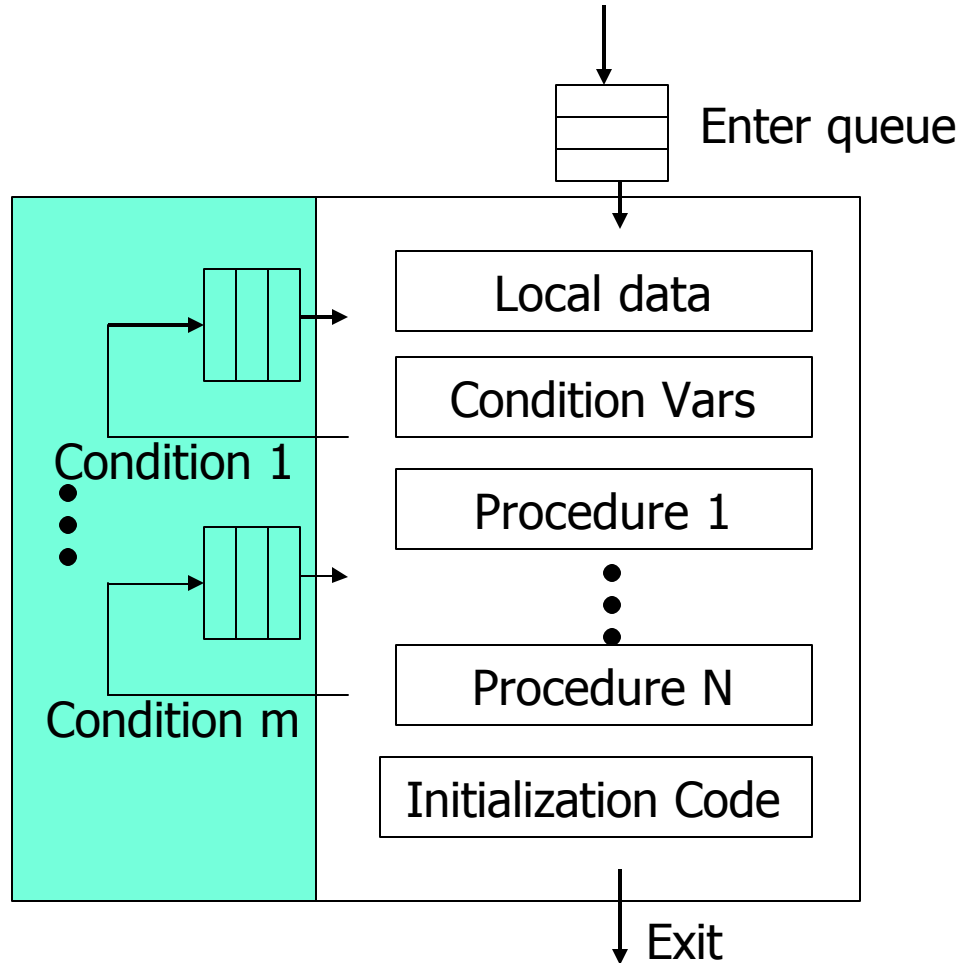
# Monitors

- an abstract data type (data and operations on the data) with the restriction that only one process at a time can be active within the monitor
  - Local data accessed only by the monitor's procedures (not by any external procedure)
  - A process enters the monitor by invoking 1 of its procedures
  - Other processes that attempt to enter monitor are blocked

# Enforcing single access

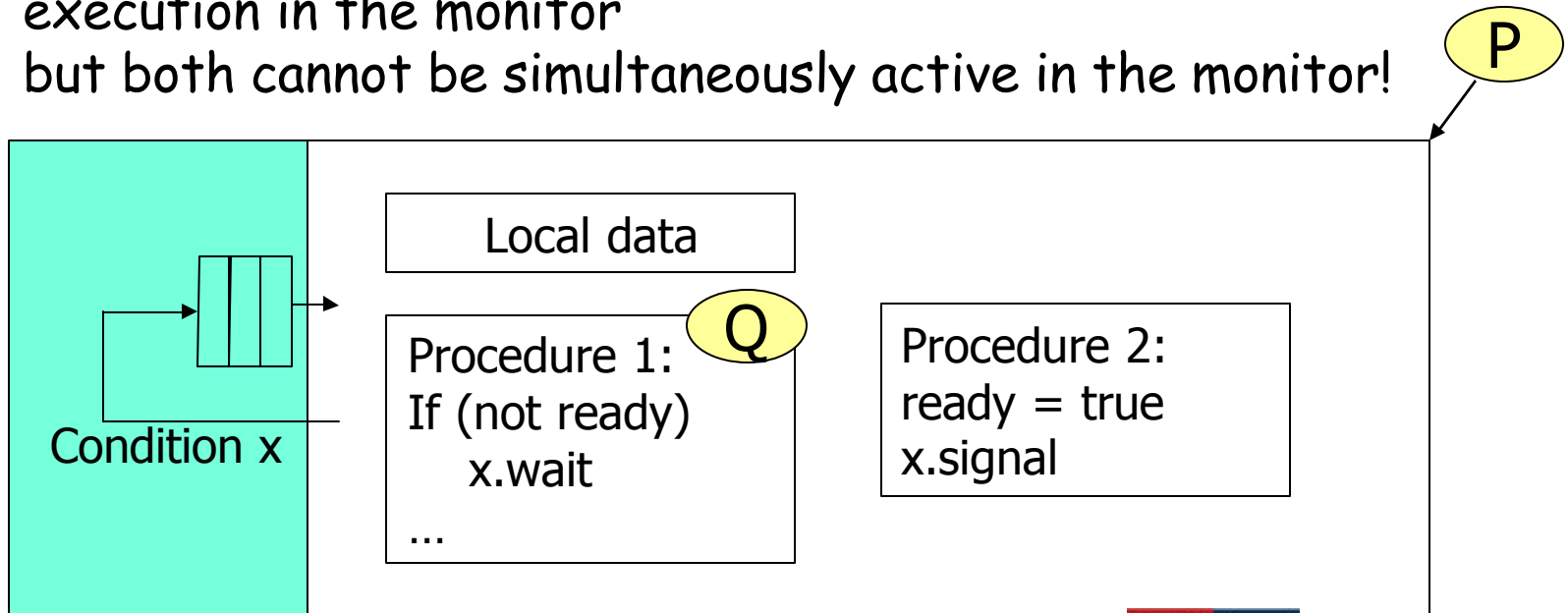
- A process in the monitor may need to wait for something to happen
  - May need to allow another process to use the monitor
  - provide a special type of variable called a condition
  - Operations on a condition variable are:
    - wait (suspend the invoking process)
    - signal (resume exactly one suspended process)
      - if no process is suspended, a signal has no effect (to be contrasted with the signal operation on semaphores, which always affects the state of the semaphore)

# Monitor Diagram



# More on Monitors

- If process  $P$  executes an  $x.\text{signal}$  operation and  $\exists$  a suspended process  $Q$  associated with condition  $x$ , then we have a problem:
  - $P$  is already "in the monitor", does not need to block
  - $Q$  becomes unblocked by the signal, and wants to resume execution in the monitor
  - but both cannot be simultaneously active in the monitor!



# Monitor Semantics for Signal

- 2 possibilities exist
  - Hoare monitors (original)
    - `signal()` immediately switches from the caller to a waiting thread
    - The condition that the waiter was blocked on is guaranteed to hold when the waiter resumes
    - Need another queue for the signaler, if signaler was not done using the monitor
  - Mesa monitors (Mesa, Java, Nachos, OS/161)
    - `Signal()` places a waiter on the ready queue, but signaler continues inside monitor
    - Condition is not necessarily true when waiter resumes
    - Must check condition again

# Hoare vs. Mesa Semantics

- Hoare
  - if (empty)
  - wait(condition);
- Mesa
  - while(empty)
  - wait(condition)
- Tradeoffs
  - Hoare monitors make it easier to reason about program
  - Mesa monitors are easier to implement, more efficient, can support additional ops like broadcast

# Using Monitors in C

- Not integrated with the language (as in Java)
- Bounded buffer: Want a monitor to control access to a buffer of limited size,  $N$ 
  - Producers add to the buffer if it is not full
  - Consumers remove from the buffer if it is not empty
- Need two functions - `add_to_buffer()` and `remove_from_buffer()`
- Need one lock - only lock holder is allowed to be active in one of the monitor's functions
- Need two conditions - one to make producers wait, one to make consumers wait

# Bounded Buffer Monitor - Variables

```
#define N 100
typedef struct buf_s {
    int data[N];
    int inpos; /* producer inserts here */
    int outpos; /* consumer removes from here */
    int numelements; /* # items in buffer */
    struct lock *buflock; /* access to monitor */
    struct cv *notFull; /* for producers to wait */
    struct cv *notEmpty; /* for consumers to wait */
} buf_t;

buf_t buffer;
void add_to_buff(int value);
int remove_from_buff();
```

# Bounded Buffer Monitor - Functions

```
void add_to_buf(int value) {
    lock_acquire(buffer.mylock);
    while (nelements == N) {
        /* buffer is full, wait */
        cv_wait(buffer.notFull, buffer.mylock);
    }
    buf.data[inpos] = value;
    inpos = (inpos + 1) % N;
    nelements++;
    cv_signal(buffer.notEmpty, buffer.mylock);
    lock_release(buffer.mylock);
}
```

# Bounded Buffer Monitor - Remove

```
int remove_from_buf() {
    int val;
    lock_acquire(buffer.mylock);
    while (nelements == 0) {
        /* buffer is empty, wait */
        cv_wait(buffer.notEmpty, buffer.mylock);
    }
    val = buf.data[outpos];
    outpos = (outpos + 1) % N;
    nelements--;
    cv_signal(buffer.notFull, buffer.mylock);
    lock_release(buffer.mylock);
}
```

# Deadlock and Starvation

- a set of threads is in a **deadlocked** state when every process in the set is waiting for an event that can be caused only by another process in the set
- in the case of semaphores, the event is the execution of the signal operation
- a thread is suffering **starvation** (or indefinite postponement) if it is waiting indefinitely because other threads are in some way preferred

# Atomic Transactions

- Recall ATM banking example:
  - Concurrent deposit/withdrawal operation
  - Need to protect shared account balance
- What about transferring funds between accounts?
  - Withdraw funds from account A
  - Deposit funds into account B

# Properties of funds transfer

- Should appear as a single operation
  - Another process reading the account balances should see either both updates, or none
- Either both operations complete, or neither does
  - Need to recover from crashes that leave transaction partially complete

# Definitions for Transactions

- Defn: Transaction
  - A collection of operations that performs a single logical function
  - We will consider a sequence of **read** and **write** operations, terminated by a **commit** or **abort**
- Defn: Committed
  - A transaction that has completed successfully; once committed, a transaction cannot be undone
- Defn: Aborted
  - A transaction that did not complete normally

# Write-ahead logging

- Before performing any operations on the data, write the intended operations to a log on stable storage
- Log records identify the transaction, the data item, the old value, and the new value
- Special records indicate the start and commit (or abort) of a transaction
- Log can be used to undo/redo the effect of any transactions, allowing recovery from arbitrary failures

# Checkpoints

- Limitations of basic log strategy:
  - Time-consuming to process entire log after failure
  - Large amount of space required by log
  - Performance penalty - each write requires a log update before the data update
- Checkpoints help with first two problems
  - Periodically write all updates to log and data to stable storage; write a checkpoint entry to the log
  - Recovery only needs to look at log since last ckpt.

# Concurrent Transactions

- Transactions must appear to execute in some arbitrary but serial order
  - Soln 1: All transactions execute in a critical section, with a single common lock (or mutex semaphore) to protect access to all shared data.
    - But most transactions will access different data
    - Limits concurrency unnecessarily

# Serializability

- Operations from different transactions can overlap, as long as they don't conflict

$T_0$	$T_1$
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

# Conflicting Operations

- Operations in two different transactions **conflict** if both access the same data item and at least one is a write
  - Non-conflicting operations can be reordered (swapped with each other) without changing the outcome
  - If a serial schedule can be obtained by swapping non-conflicting operations, then the original schedule is **conflict-serializable**

# Ensuring serializability

- Two-phase locking
  - Individual data items have their own locks
  - Each transaction has a **growing** phase and **shrinking** phase:
    - Growing: a transaction may obtain locks, but may not release any lock
    - Shrinking: a transaction may release locks, but may not acquire any new locks.
  - Does not guarantee deadlock-free

# Timestamp Protocols

- Each transaction gets unique **timestamp** before it starts executing
  - Transaction with "earlier" timestamp must appear to complete before any later transactions
- Each data item has two timestamps
  - W-TS: the largest timestamp of any transaction that successfully wrote the item
  - R-TS: the largest timestamp of any transaction that successfully read the item

# Timestamp Ordering

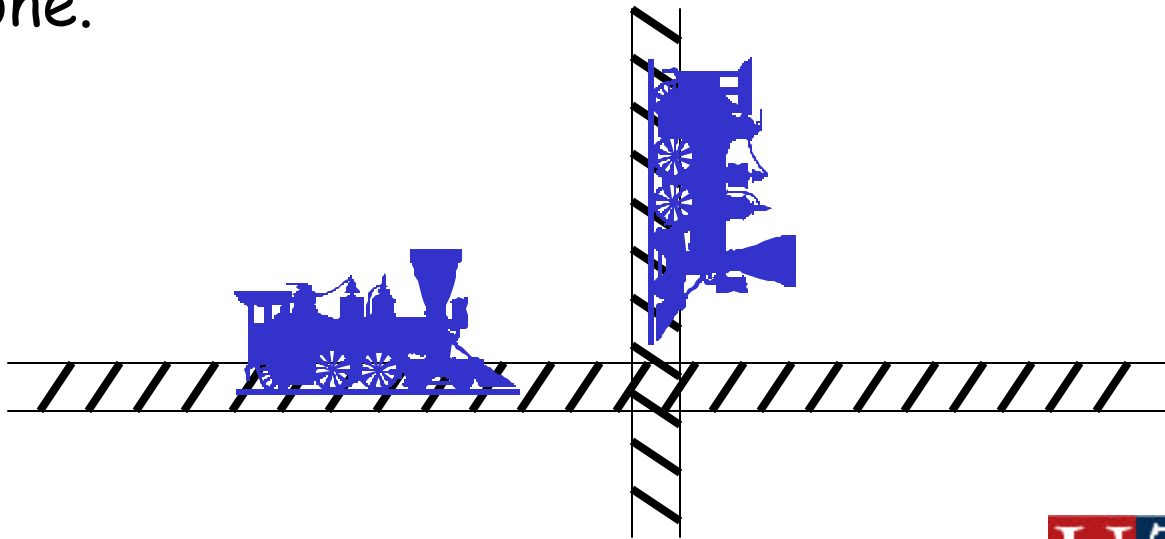
- Reads:
  - If transaction has "earlier" timestamp than W-TS on data, then transaction needs to read a value that was already overwritten
    - Abort transaction, restart with new timestamp
- Writes:
  - If transaction has "earlier" timestamp than R-TS (W-TS) on data, then the value produced by this write should have been read (overwritten) already!
    - Abort & restart
- Some transactions may "starve" (abort & restart repeatedly)

# Deadlock Defined

- The permanent blocking of a set of processes that either:
  - Compete for system resources, or
  - Communicate with each other
- Each process in the set is blocked, waiting for an event which can only be caused by another process in the set
  - Resources are finite
  - Processes wait if a resource they need is unavailable
  - Resources may be held by other waiting processes

# Not just an OS Problem!

- Law passed by Kansas Legislature in early 20th Century:
  - "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start upon again until the other has gone."

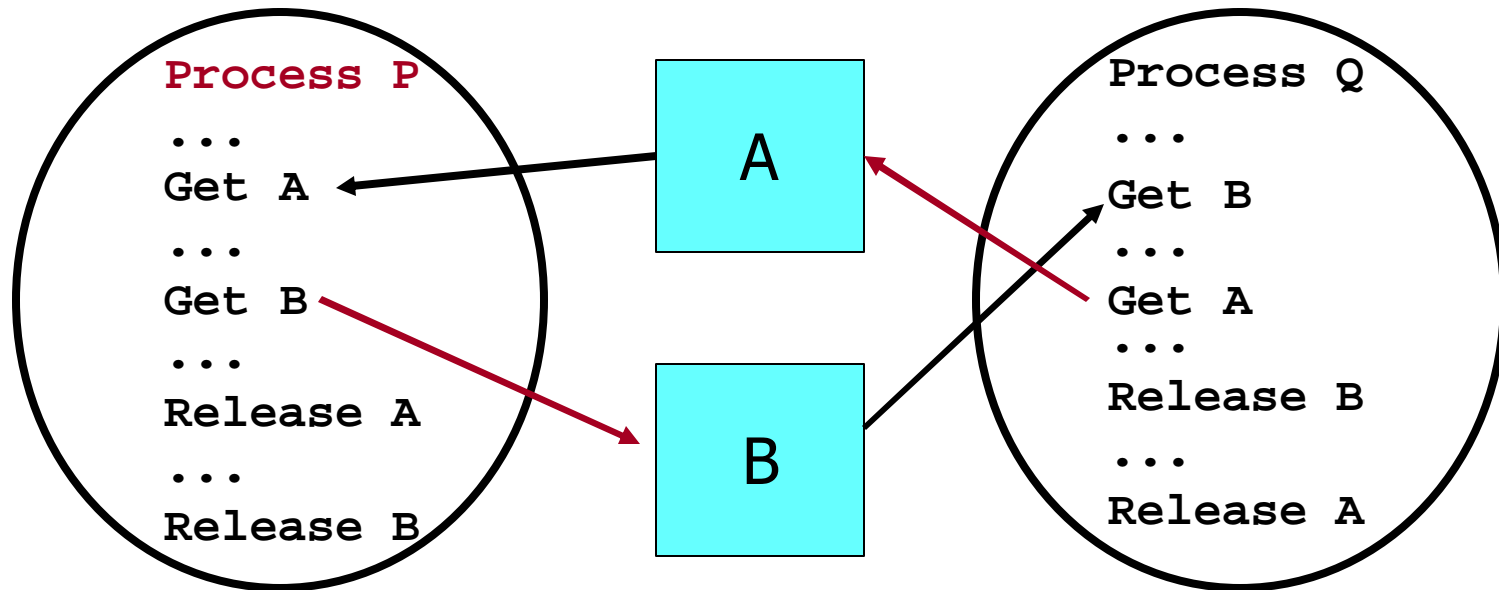


# Types of Resources

- Reusable
  - Can be used by one process at a time, released and used by another process
    - printers, memory, processors, files
    - Locks, semaphores, monitors
- Consumable
  - Dynamically created and destroyed
  - Can only be allocated once
    - e.g. interrupts, signals, messages

# Example of Deadlock

- Suppose processes **P** and **Q** need (reusable) resources **A** and **B**:



# Conditions for Deadlock

## 1. Mutual Exclusion

- Only one process may use a resource at a time

## 2. Hold and wait

- A process may hold allocated resources while awaiting assignment of others

## 3. No preemption

- No resource can be forcibly removed from a process holding it
- These are **necessary** conditions

# One more condition...

## 4. Circular wait

- A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain
- Together, these four conditions are **necessary and sufficient** for deadlock
- Circular wait implies hold and wait, but the first results from a sequence of events, while the second is a policy decision

# Deadlock Prevention

- Ensure one of the four conditions doesn't occur
  - Break mutual exclusion - not much help here, as it is often required for correctness
    - Neil Groundwater about Unix at Bell Labs in 1972:
      - “... There was no spooling or lockout [for the line printer]. `pr myfile > /dev/lp` was how you sent your listing to the printer. If two users sent output to the printer at the same time, their outputs were interspersed. Whoever shouted 'line printer!' first owned the queue.”

# Preventing Hold-and-Wait

- Break "hold and wait" - processes must request all resources at once, and will block until entire request can be granted simultaneously
  - May wait a long time for all resources to be available at the same time
  - May hold resources for a long time without using them (blocking other processes)
  - May not know all resource requirements in advance
- An alternative is to release all currently-held resources when a new one is needed, then make a request for the entire set of resources

# Preventing No-Preemption

- Break “no preemption” - forcibly remove a resource from one process and assign it to another
  - Need to save the state of the process losing the resource so it can recover later
  - May need to rollback to an earlier state
  - Name some resources that this works for...
  - Name some resources for which this is hard...
  - Impossible for consumable resources

# Preventing Circular-wait

- Break "circular wait" - assign a linear ordering to resource types and require that a process holding a resource of one type,  $R$ , can only request resources that follow  $R$  in the ordering
  - e.g.  $R_i$  precedes  $R_j$  if  $i < j$
  - For deadlock to occur, need  $P$  to hold  $R_i$  and request  $R_j$ , while  $Q$  holds  $R_j$  and requests  $R_i$
  - This implies that  $i < j$  (for  $P$ 's request order) and  $j < i$  (for  $Q$ 's request order), which is impossible.
- Hard to come up with total order when there are lots of resource types

# Deadlock Avoidance

- All prevention strategies are unsatisfactory in some situations
- Avoidance allows the first three conditions, but orders events to ensure circular wait does not occur
  - How is this different from preventing circular wait?
- Requires knowledge of future resource requests to decide what order to choose
  - Amount and type of information varies by algorithm

# Two Avoidance Strategies

1. Do not start a process if its maximum resource requirements, together with the maximum needs of all processes already running, exceed the total system resources
  - Pessimistic, assumes all processes will need all their resources at the same time
2. Do not grant an individual resource request if it might lead to deadlock

# Safe States

- A state is **safe** if there is at least one sequence of process executions that does not lead to deadlock, even if every process requests their maximum allocation immediately
- Example: 3 processes, 1 resource type, 10 instances

T0: Available = 3	PID	Alloc	Max Claim
T1: Available = 1	A	3	9
T2: Available = 5	B	<del>2</del> <del>4</del> 0	4
T3: Available = 0	C	<del>2</del> <del>7</del> 0	7
T4: Available = 7			

# Unsafe States & Algorithm

- An **unsafe** state is one which is not safe
  - Is this the same as a deadlocked state?
- Deadlock avoidance algorithm
  - For every resource request
    - Update state assuming request is granted
    - Check if new state is safe
    - If so, continue
    - If not, restore the old state and block the process until it is safe to grant the request
- This is the **banker's algorithm**
  - Processes must declare maximum needs
  - See text for details of the algorithm

# Restrictions on Avoidance

- Maximum resource requirements for each process must be known in advance
- Processes must be independent
  - If order of execution is constrained by synchronization requirements, system is not free to choose a safe sequence
- There must be a fixed number of resources to allocate
  - Tough luck if a printer goes offline!

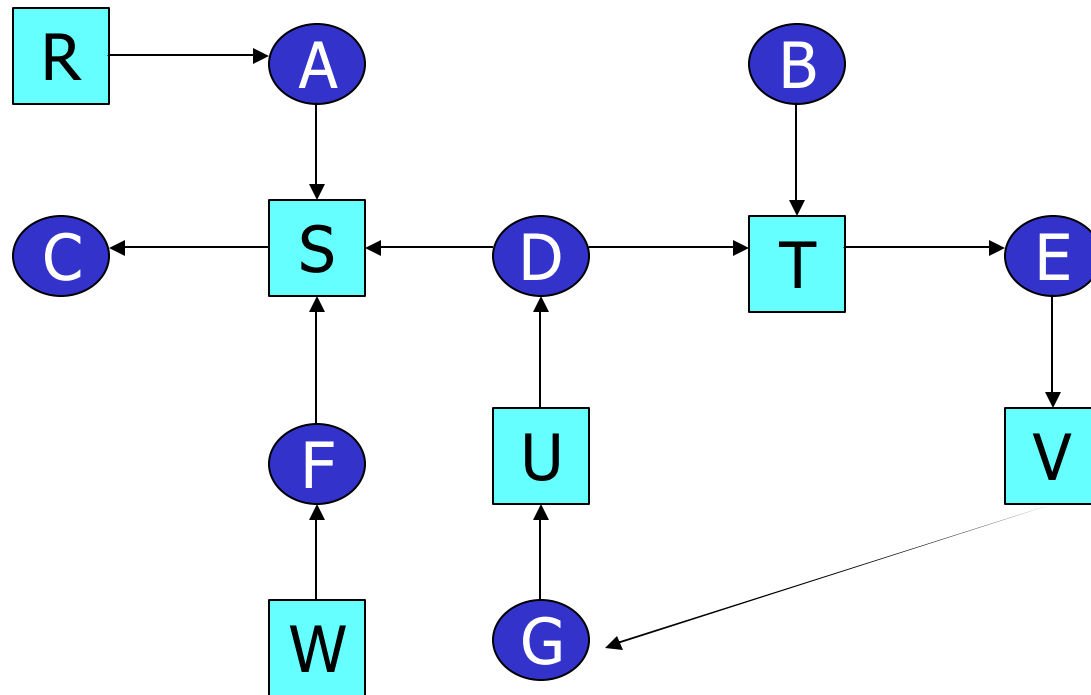
# Deadlock Detection & Recovery

- Prevention and avoidance is awkward and costly
  - Need to be cautious, thus low utilization
- Instead, allow deadlocks to occur, but detect when this happens and find a way to break it
  - Check for circular wait condition periodically
- When should the system check for deadlocks?

# Detection continued

- Finding circular waits is equivalent to finding a cycle in the resource allocation graph
  - **Nodes** are **processes** (drawn as circles) and **resources** (drawn as squares)
  - **Arcs** from a resource to a process represent **allocations**
  - **Arcs** from a process to a resource represent **ungranted requests**
- Any algorithm for finding a cycle in a directed graph will do
  - note that with multiple instances of a type of resource, cycles may exist without deadlock

# Example Resource Alloc Graph



# Deadlock Recovery

- Basic idea is to break the cycle
  - Drastic - kill all deadlocked processes
  - Painful - back up and restart deadlocked processes (hopefully, non-determinism will keep deadlock from repeating)
  - Better - selectively kill deadlocked processes until cycle is broken
    - Re-run detection alg. after each kill
  - Tricky - selectively preempt resources until cycle is broken
    - Processes must be rolled back

# Reality Check

- No single strategy for dealing with deadlock is appropriate for all resources in all situations
- All strategies are costly in terms of computation overhead, or restricting use of resources
- Most operating systems employ the "Ostrich Algorithm"
  - Ignore the problem and hope it doesn't happen often

# Why does the Ostrich Alg Work?

- Recall causes of deadlock:
  - Resources are finite
  - Processes wait if a resource they need is unavailable
  - Resources may be held by other waiting processes
- Prevention/Avoidance/Detection mostly deal with last 2 points
- Modern operating systems **virtualize** most physical resources, eliminating the first problem
  - Some logical resources can't be virtualized (there has to be exactly one), such as bank accounts or the process table
    - These are protected by synchronization objects, which are now the only resources that we can deadlock on