**EXAM NOTES - CS2850 Operating Systems**

**Mutual Exclusion** = technique to avoid race condition

1 process use, other process are excluded

Identify parts that access shared resources **critical region** and prevent more than 1 process from access

**Critical Region**

1. **No 2 process** can simultaneously inside
2. **No assumptions** made on speed/no. of CPUs
3. No process outside CR may **block other** process
4. No process should **wait forever** (CR)

**BUSY WAIT**

= 1 proc is busy updating shared mem in CR, no other proc will enter CR and cause trouble. Will keep waiting

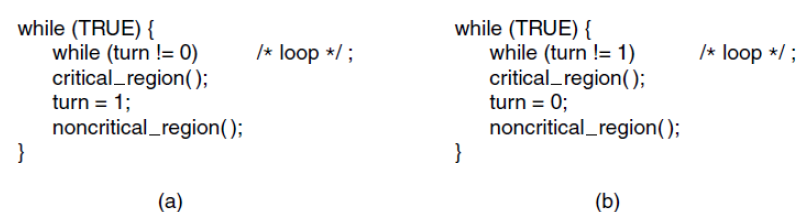
**Lock Variable**

* Variable set to 0
* Locks it (change to 1) Unlock (change to 0)

Problem – read and write are diff instructions (can be interrupted)

**Strict Alternation**

“turn” variable

****

Problems

Only applicable when wait is short

<a> is much slower than (b), (b) waits a long time

🗶 “No Proc running outside CR may block other proc”

**Peterson’s Solution**

Idea: indicates interest, but offer others to go first

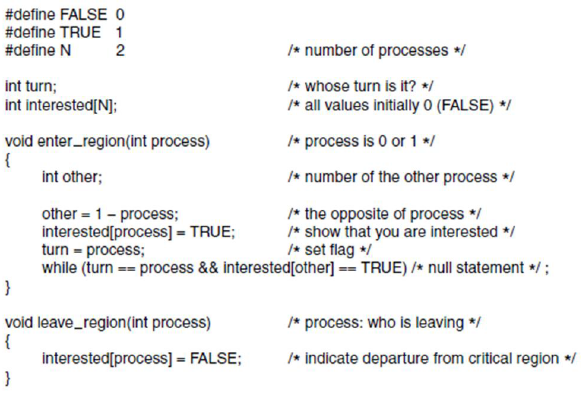
CR Control:

Enter\_region(process);

Critical\_region();

Leave\_region(process);

Before entering, proc calls with its own no. (0 or 1)

****

**Proof of mutual exclusion (by contradiction)**- P0 entering, when P1 is already   
(interested[other] = true, so P0 can only enter if turn = 1, but P0 just change turn = 0.)

-P1 entered when turn = 1, P0 is entering   
(interested[0] = true before setting turn = 0 , so P1 should be waiting in a loop)

**Proof of No Mutual Blocking**

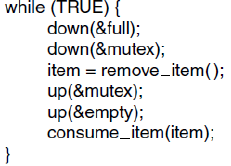
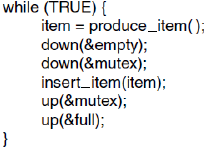
* Impossible, cause turn is either 1 or 0

(will become false, one will enter)

**Condition 4 (wait forever)**

* 1 process wait forever (when leaving, it sets interested[other] to FALSE, the other one can enter)
* Turn = offer turn to another process

**Solution: Semaphores = special variable**

* Keep info about no of wakeup calls which not taken into account
* Allow 2 main ops
  + Down – if 0 = sleep, or else decrement and allow execution
  + Up – increment (if got process sleeping due to down, 1 will be awaken to execute)
* Check value with atomic operation
* Once started, no other process can access
* Implemented as sys call:
  + OS disables interrupt
  + Multiple CPU, protect semaphore lock var with TSL/XCHG
  + Some busy wait, BUT **limited for very short semaphore calls instead of whole CR**
* 
* Used for Mutual Exclusion
  + Protect a CR, preventing race condition
* Used for synchronization
  + Prevent/force a certain seq of events
* Binary Semaphores = 2 processes only

**Scheduling in Batch Sys**

* FCFS
  + Queue
  + Not efficient
* Shortest Job first
  + If you know hw much time
  + Shortest turnaround time
  + Optimal = if available simultaneously



**Deadlocks**

Resources used by 1 proc at a time, OS needs to grant access

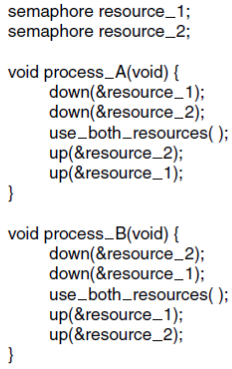
**Preemptable/nonpreemptable resources**

**P** = can be taken away temporarily

**NP =** cannot be taken away temporarily or will fail (burn DVD)

Semaphores to protect resources

Potential Deadlock (locking resources in different order)



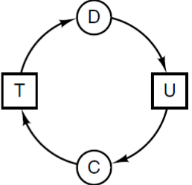
**Set of processes = deadlocked IF:**

* Each process waiting for event
* Event can only be caused another process in set

**4 conditions for deadlock**

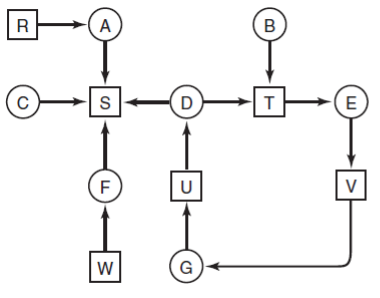
* **Mutual exclusion**
* **Hold and wait (hold R and waiting for new R)**
* **No preemption**
* **Circular wait condition**

**Deadlock Modelling**

Process = circle , square = resource

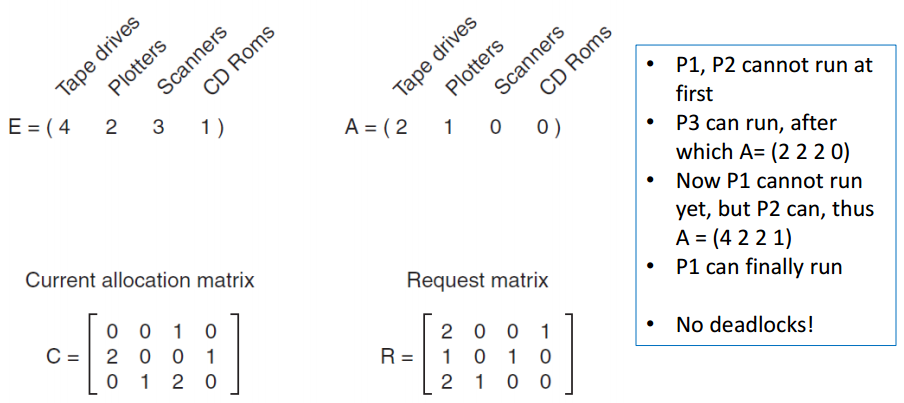
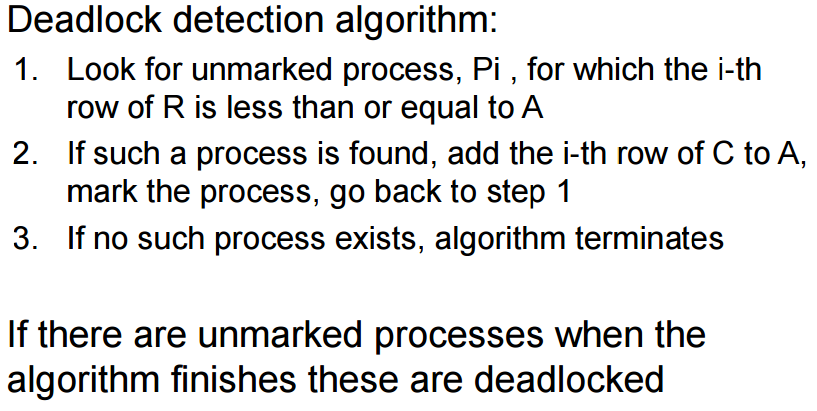
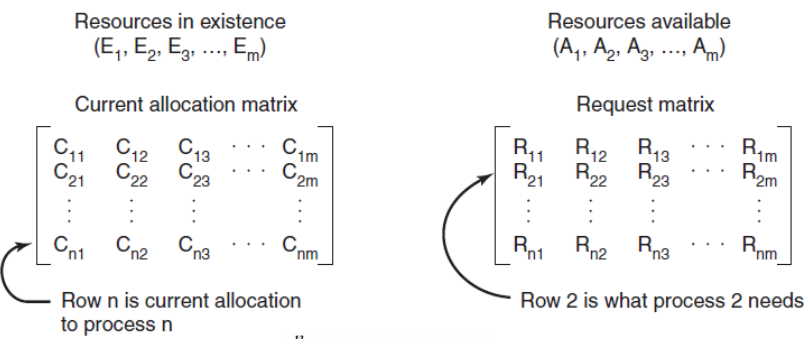
**Strategies**

1. Detection and Recovery
2. Dynamic avoidance be careful resource allocation
3. Prevention, negating 1 of 4 conditions

**Detection and Recovery**

Algo

For all nodes, follow all the nodes movement and see it if goes back to itself (detect cycles)

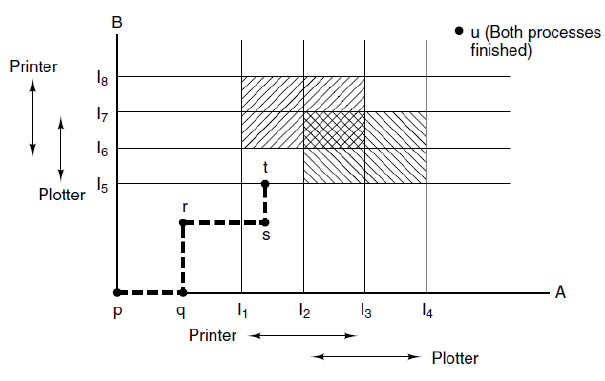
Detection for Multiple R of Each Type

**Recovery**

1. **Preemption** (dependent of nature of resource)
2. **Rollback** (save state, RB to state bef. Deadlock)
3. **Killing proc** (break the deadlock)

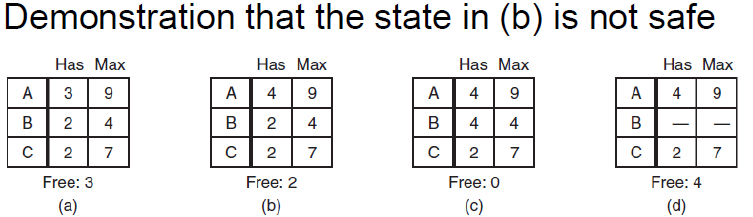
**Deadlock Avoidance**

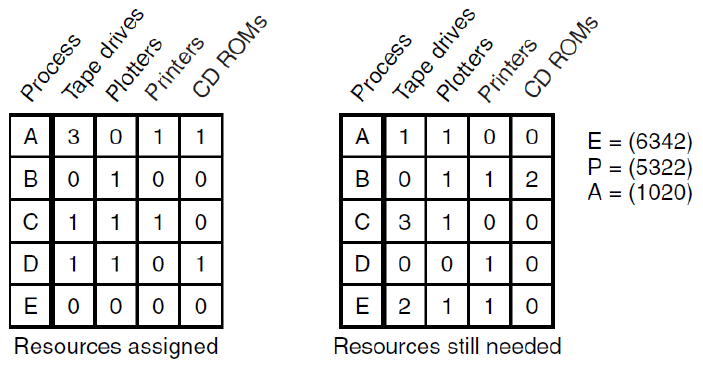
**Resource trajectories**



**Safe and Unsafe States**

System can guarantee that all proc will finish



**Banker’s Algorithm for multiple resource**

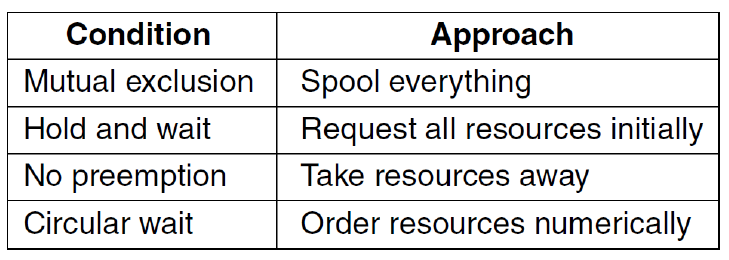
**\*diff =** has the “max resources” that the proc will req, so can know if safe or unsafe state

**Deadlock Prevention**

**Deadlock avoidance** = only if detailed information about future (not common)

**Deadlock prevention** = more realistic

* Ensure 1 of 4 deadlock conditions never satisfied
* Mutual exclusion – avoiding assigning resource unless absolutely necessary
* Hold-wait = requires all proc to request all resources before starting execution (but many proc do not know how many resources they need at start)
* No-Preemption = not all fit that category (db, needs to be locked to be used)
* Circular wait
  + Request need to be made in numerical order (then no cycle)



Spool everything = printer spooler (queue)

**Communication Deadlocks**

* Resource deadlock in **network**
* 

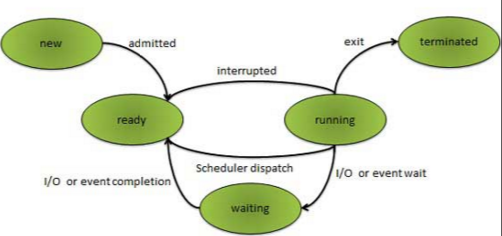
**LiveLock**

* Processes not blocked, but no progress
* 

**Starvation**

* Never have chance to use resource
* Solved by algo (1st come 1st served)

**Processes**



**Definition**

Critical Section – shared resources that only allow 1 process to access at a time

Starvation – continue to run, but no progress made

Deadlock – above

Batch – no user interaction (throughput focus)

Interactive – response time, user expectation

Realtime – multimedia, predictability

Interleaving of concurrent execution – process of managing processes and resources in a way that processes run like it is running concurrently.

**P1P2**

**Computer Architecture**

* CPU
* Main memory
* I/O

**First Fit**: find first hole that is big enough

Broken into 2 pieces, 1 for process, 1 for unused memory

**Next fit**: Scan from last segment where it found a hole

(worse performance based on simulations)

**Best fit**: Smallest hole that fit (scans whole list, slower than first fit)

(slower and waste memory, a lot of small tiny holes)

**Worst fit**: Largest hole that fit (not good also)

* Algo need to scan holes

**Quick fit:** separate list for more common sizes

* Merging = hard
* Uses added time to move segments between lists when process terminates or swap

**Page Replacement (page fault)**

Optimal Algo

* Page with highest label to removed
* Each labeled with no. of instruction until ref.
* UNFEASIBLE = can’t predict future ref.
* Used for benchmarking.

Not recently Used Algo

* System inspects pages (Ref, Mod)
* NRU with the lowest class
* Class 0: Not ref, Not mod, .., Class 3: ref, mod

FIFO Algo

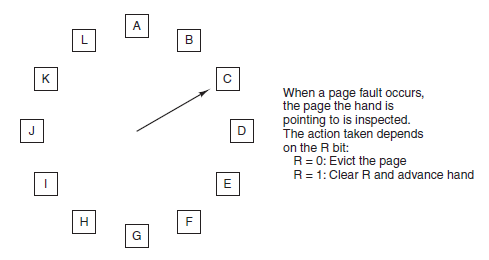
* First entry memory, first exit
* New page added at end of queue
* Problem = oldest page still useful

Second Chance Algo

* Sorted in FIFO
* If useful, queue again in FIFO

Clock Page Replacement Algo

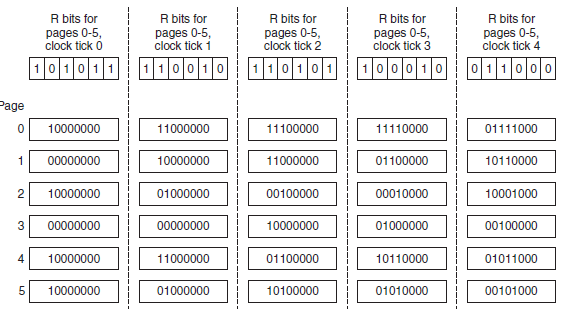
* More Efficient than 2nd chance algo, does not move pages
* Move arrow (like clock), has R value to show can evict or not



**Least Recently Used Algo**

Throw away page that has been unused for longest (based on observation)

Expensive = needs all pages and access time

Simulating LRU Algo = Aging

**Working Set** (pages that a process is currently using)

- Purest form, no pages in memory

- **Demand paging** = fetches all the stuff required to run, settles down and few page faults after some time

**Locality of reference** (at any phase, process references small fraction of pages)

Paging system can load the current working set once a process gets swapped back into physical memory **(working set model, prepaging)**