

Operating Systems & Deadlocks

Concurrency, Process synchronization

- single process system $\neq C$
- fixed time slot allocation $= C$
- context switch

Inter-process Communication

- Requirements: 1) Synchronization
- 2) Communication

Synchronization of process \rightarrow mutual exclusion

Independent process \neq communication

Cooperative process communicate

With shared mem. or msg passing

Shared memory

Msg passing \rightarrow send (send, msg)

receive (recv, msg)

Process / Thread Synchronization

- Critical Section Pb

portion of the pg where some mem. is shared \rightarrow critical section

Properties

- read - update - write (both)

Conditions:

- 1) mutual exclusion
- 2) progress

3) bounded waiting

Rate Condition

- undesirable situation \rightarrow when a system attempts to perform more operations at same time but because of nature of system, ops must be done in proper sequence

Rule of OS

- keeping track of processes
- allocation & reclamation
- security
- process interaction

Peterson's Pb

do {
flag[i] = true;
turn = i;
while (flag[j] & turn == j) continue;
/* critical section */
}

Mutual Exclusion

- deadlock condition
- holding true for non sharable resource (time, memory, space)

- significance

- 1) prevents race around condition
- 2) prevents multiple thread to enter CS at same time

Requirements:

- deadlock \rightarrow endless waiting

- starvation \rightarrow unbounded waiting
- unfairness \rightarrow considered, un-used
- fault intolerance

Hardware Support

- make programming tasks easier
- get better system efficiency
- to/w/o OS maintenance

Operating System Support

- Software approach for mutual exclusion
- E.M. Dijkstra (1965) \rightarrow Semaphore
- C.S. not easy to generalize

To avoid complexity \rightarrow use a synchronized but \rightarrow semaphore

- Semaphore: \rightarrow counting
- value is designated place in OS/mem. where each process can check from change

- Peterson's solution

- good algorithm concept of using C.S.P
- mutual exclusion preserved
- progress requirement satisfied
- bounded waiting requirement is not

Disadv:

- waste of CPU time
- 2 processes involved

Flag 1: False

/* remainder section */
while (true) {
}

Programming Language Support (Monitor)

- If semaphores are used incorrectly
- \Rightarrow Timing errors \rightarrow hard to detect
- to handle multi user high level language developed \rightarrow monitors

Monitors \rightarrow set of procedures, variables & data structures that are grouped together in package

eg: semaphore monitor

data structure: procedure proc()

procedure proc() {
}

procedure proc() {
}

procedure proc() {
}

procedure proc() {
}

procedure proc() {
}

procedure proc() {
}

procedure proc() {
}

procedure proc() {
}

procedure proc() {
}

procedure proc() {
}

procedure proc() {
}

Deadlock

- in multi-programming environment many processes compete for multiple resources
- in system \rightarrow hold resources

- with finite resources that are held by processes

- waiting state can remain in same state & will never acquire change state

- Sequence \rightarrow request \rightarrow use \rightarrow release

Deadlock Condition

- process never finish execution, no resources used by other processes
- close to that resources in which some process are scheduled for execution
- Mutual exclusion
- hold and wait
- no preemption
- Circular wait

Deadlock Prevention

- preventing by by constraining how resources for resources can be made in system & how they are handled by system
- goal is to ensure that at least one of the necessary condition for it can now hold

- Deadlocks Pb

- Producer Pb

- Consumer Pb

- Producer Pb

- Consumer Pb

- Producer Pb

- Consumer Pb

- Producer Pb

- Consumer Pb

Resource allocation graph

- pictorial representation of state of system to keep complete info about all processes & instances
- in RAG, process $\rightarrow O$, resource $\rightarrow \Delta$

eg:

Deadlock Assignment

- It requires that the system has some info about system

Deadlock Algorithm

- 1) Resource allocation graph algorithm
- 2) Banker's algorithm
- 3) Resource request algorithm
- 4) Safety algorithm

Deadlock Detection

- process of determining that all extra resources involved in the system are available for all possible allocation
- if system is in the state

Deadlock Recovery

- remove from the system, if detected
- to inform operating system how decision have to deal with it manually

Dynamic (Heuristic) Pb (1977)

- wait (change state)
- wait (change state)
- wait (change state)

Signal (change state)

- Signal (change state)

Signal (change state)

- Signal (change state)

Signal (change state)

- Signal (change state)

Signal (change state)

- Signal (change state)

Signal (change state)

- Signal (change state)

Signal (change state)

- Signal (change state)

Signal (change state)

- Signal (change state)

Signal (change state)

- Signal (change state)

Need Matrix

Process Allocation max Available

Process	A	B	C	D	A	B	C	D
P1	0	0	0	0	1	5	2	0
P2	1	0	0	0	1	5	2	0
P3	1	3	5	2	3	5	2	0
P4	0	6	3	0	6	5	2	0
P5	0	1	4	0	6	5	2	0

Need Matrix \Rightarrow Need + Max - Allocation

Process A B C D

Process	A	B	C	D
P1	0	0	0	0
P2	0	7	5	0
P3	1	0	0	2
P4	0	0	2	0
P5	0	6	4	2

Need \leq available

P1 \rightarrow 0000 \leq 1520 \rightarrow True Satisfied

P2 \rightarrow 0750 \leq 1520 \rightarrow False NS

P3 \rightarrow 1002 \leq 1520 \rightarrow True S

P4 \rightarrow 0020 \leq 1520 \rightarrow True S

P5 \rightarrow 0642 \leq 1520 \rightarrow True S

P1 \rightarrow 0750 \leq 1520 \rightarrow False NS

P2 \rightarrow 0750 \leq 1520 \rightarrow False NS

P3 \rightarrow 1002 \leq 1520 \rightarrow True S

P4 \rightarrow 0020 \leq 1520 \rightarrow True S

P5 \rightarrow 0642 \leq 1520 \rightarrow True S

P1 \rightarrow 0750 \leq 1520 \rightarrow False NS

P2 \rightarrow 0750 \leq 1520 \rightarrow False NS

P3 \rightarrow 1002 \leq 1520 \rightarrow True S

P4 \rightarrow 0020 \leq 1520 \rightarrow True S

P5 \rightarrow 0642 \leq 1520 \rightarrow True S

Safe Sequence

$\langle P_0, P_1, P_2, P_4, P_7 \rangle$

Addition request

P1 = $\langle 0, 4, 2, 0 \rangle$

R \leq N

0420 \leq 0750

R \leq AV

0420 \leq 1520

AV = AV - Request

= 1520 - 0420

= 1100

Allocation: All + Req

= 1000 + 0420

= 1420

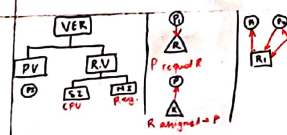
Need = Need - Request

= 0750 - 0420

= 0330

2.1 \rightarrow New need matrix

2.2 \rightarrow New matrix

<h3>Synchronization & Deadlocks</h3> <p>Concurrency / Process synchronization</p> <ul style="list-style-type: none"> - single process system $\neq C$. - fixed time slot allocation $= C$ - context switch <p>Interprocess Communication</p> <ul style="list-style-type: none"> - Requirements: 1) Synchronization 2) Communication <p>- Synchronization of process \rightarrow mutual exclusion</p> <p>- Independent process \neq communicates</p> <p>- Cooperative processes communicate with shared mem. or msg passing.</p> <p>Shared memory</p> <p>Msg passing $\left[\begin{array}{l} \text{send (dst, msg)} \\ \text{receive (src, msg)} \end{array} \right.$</p> <p>Process / Thread Synchronization</p> <ul style="list-style-type: none"> • Critical Section Pb <p>- portion of the pg where shared mem. is accessed \rightarrow critical section</p> <p>Properties</p> <ul style="list-style-type: none"> - read - update - write (modify) <p>Conditions:</p> <ol style="list-style-type: none"> 1) mutual exclusion 2) progress 3) bounded waiting <p>• Race Condition</p> <p>- undesirable situation \rightarrow when a system attempts to perform 2 or more operations at same time but because of nature of system, ops must be done in proper sequence</p> <p>Role of OS</p> <ul style="list-style-type: none"> - Keeping track of processes. - allocation & reclamation - security - process interaction <p>Peterson's Pb</p> <pre> do { flag[i] = true; turn = i; while (flag[j] & turn == j) /* critical section */ } </pre>	<p>Mutual Exclusion</p> <ul style="list-style-type: none"> - deadlock condition - holding true for non sharable resources <small>printer, memory space</small> <ul style="list-style-type: none"> • Significance <ul style="list-style-type: none"> 1) prevents race around condition 2) prevents multiple thread to enter CS at same time • Requirements: <ul style="list-style-type: none"> - deadlock \rightarrow endless waiting <ul style="list-style-type: none"> - starvation \rightarrow unbounded waiting - unfairness \rightarrow unordered, unsorted requests - fault intolerance <p>Hardware Support</p> <ul style="list-style-type: none"> - make programming tasks easier - gets better system efficiency - h/w & sw maintenance 	<p>Programming Language Support (Monitors)</p> <ul style="list-style-type: none"> - If semoph are used incorrectly \Rightarrow Timing errors \rightarrow hard to debug - to handle multi error, high level language <p>developed \rightarrow monitors</p> <p>Monitors \rightarrow set of procedures variable & data structures that are grouped together in packages</p> <p>Eg: monitor monitorName;</p> <p>Functions:</p> <ol style="list-style-type: none"> 1) wait(): <ul style="list-style-type: none"> • Execution of calling process get point • After this other process can access the monitor 2) signal(): <p>data variables; procedure P1(...); procedure P2(...); procedure Pn(...); Initialization Code(...);</p>	<p>Deadlock</p> <ul style="list-style-type: none"> - in multi programming environment many processes compete for multiple resources - in system \rightarrow finite resources - with finite resources not possible to fulfill resource request of all process - waiting state can remain in same state & will never again change state - Sequence \rightarrow request \rightarrow use \rightarrow release <p>Deadlock Condition</p> <ul style="list-style-type: none"> - process never finish execution as resources used by other processes - due to this unavailability \rightarrow new process are prevented from starting execution <p> <input checked="" type="checkbox"/> Mutual Exclusion <input checked="" type="checkbox"/> Hold and wait <input checked="" type="checkbox"/> No preemption <input checked="" type="checkbox"/> Circular Wait </p>	<p>Resource allocation graph</p> <ul style="list-style-type: none"> - pictorial representation \rightarrow state of system - keeps complete info about all processes & instances - in RAG, process \rightarrow O, resource \rightarrow R  <p>Deadlock Avoidance</p> <ul style="list-style-type: none"> - It requires that the system has some info. avail. upfront <p>Initially, max no. of resources need which it may require</p> <p>Deadlock Algorithm</p> <ol style="list-style-type: none"> 1) Resource allocation graph algorithm 2) Banker's algorithm 3) Resource-request algorithm 4) Safety algorithm
<p>Operating System Support</p> <ul style="list-style-type: none"> • Software approach for mutual exclusion - E.W. Dijkstra (1965) \rightarrow Key notion of m.e \rightarrow semaphores - C.S. not easy to generalize <p>To avoid complexity \rightarrow use a <u>synchronized</u> b/w \rightarrow semaphores</p> <p>Semaphores $\left[\begin{array}{l} \text{Binary} \\ \text{counting} \end{array} \right.$</p> <ul style="list-style-type: none"> - value in designated place in OS/hw. Storage that each process can check then change <p>• Peterson's Solution</p> <ul style="list-style-type: none"> - good algorithmic descript of solving C.S.P - mutual exclusion preserved - progress requirement satisfied - bounded waiting requirement is met <p>Disadv:</p> <ul style="list-style-type: none"> - waste of CPU time - 2 processes involved 	<p>Synchronization Pb - Reader/Writer's Pb</p> <ul style="list-style-type: none"> • Reader's Pb <pre> wait(mutex); rc++; if (rc == 1) wait(wrt); Signal(mutex); // Read the object wait(mutex); rc--; if (rc == 0) Signal(wrt); Signal(mutex); </pre> <ul style="list-style-type: none"> • Writer's Pb <pre> wait(wrt); // Write into the object Signal(wrt); </pre> <p>Procedure & Consumer's Pb.</p> <ul style="list-style-type: none"> - in this, buffer \rightarrow bounded buffer - finite no. of slots available in buffer - While producing, buffer \rightarrow full producer process - While consuming, buffer \rightarrow full consumer process - To avoid race condit, sleep-wake up system call used - When buffer full, producer \rightarrow sleep, consumer \rightarrow wake - When buffer empty, producer \rightarrow wake up, consumer \rightarrow sleep <p>- P & C share a common fixed size buffer</p> <ul style="list-style-type: none"> - When buffer already full \rightarrow difficulties for procedure, procedure \rightarrow sleep, consumer \rightarrow wake \rightarrow remove data - When buffer empty \rightarrow difficulties to consume - consumer \rightarrow sleep, procedure \rightarrow wakes up - when puts some data 	<p>Deadlock Prevention</p> <ul style="list-style-type: none"> - preventing DL by constraining how requests for resources can be made in system & how they are handled by system - Goal is to ensure that at least one of the necessary condition for DL can never hold <ul style="list-style-type: none"> • Producer's Pb <pre> do { // Produce item wait(empty); wait(mutex); // Put item in buffer Signal(mutex); Signal(full); } while (1); </pre> <ul style="list-style-type: none"> • Consumer's Pb <pre> do { wait(full); wait(mutex); // Remove item from buffer Signal(mutex); Signal(empty); // Consume item } while (1); </pre> <p>Difficultly of Sol:</p> <ul style="list-style-type: none"> - No a neighbouring ps can eat at the same time - If all ps pick one chopstick at the same time then deadlock occurs - Ways to avoid deadlock - At most 4 philosophers should eat 	<p>Deadlock Detection</p> <ul style="list-style-type: none"> - process of determining that DL exists & resources involved in DL - idea \rightarrow check allocation availability for all possible allocation to determine if system is in DL state <p>Deadlock Recovery</p> <ul style="list-style-type: none"> - recover from DL when it's detected - to inform operator & let them decide how to deal with it manually <p>Dining Philosopher's Pb (DPP)</p> <p>• Code</p> <pre> do { wait(chopstick[i]); wait(chopstick[(i+1)%5]); // EATING THE RICE Signal(chopstick[i]); Signal(chopstick[(i+1)%5]); } while (1); </pre> <p>Resources & Shared Memory</p>	<p>Difficultly of Sol:</p> <ul style="list-style-type: none"> - No a neighbouring ps can eat at the same time - If all ps pick one chopstick at the same time then deadlock occurs - Ways to avoid deadlock - At most 4 philosophers should eat <p>Even ps \rightarrow Right chopstick then left chopstick</p> <p>Odd ps \rightarrow Left chopstick then Right chopstick</p> <p>Ps should only be allowed to pick up if both chopsticks are available</p>

Numerical

Process	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P ₀	0	0	1	2	0	0	1	2	1	5	2	0
P ₁	1	0	0	0	1	7	5	0				
P ₂	1	3	5	4	2	3	5	6				
P ₃	0	6	3	2	0	6	5	2				
P ₄	0	0	1	4	0	6	5	6				

① Need Matrix \Rightarrow Need = Max - Allocation

Process	A	B	C	D
P ₀	0	0	0	0
P ₁	0	7	5	0
P ₂	1	0	0	2
P ₃	0	0	2	0
P ₄	0	6	4	2

Need \leq available

② Safe Sequence 4

$\langle P_0, P_2, P_3, P_4, P_1 \rangle$

③ Addition request
P₁ = (0 4 2 0)

$$R \leq N$$

$$0 \ 4 \ 2 \ 0 \leq 0 \ 7 \ 5 \ 0$$

$$R \leq av$$

$$0 \ 4 \ 2 \ 0 \leq 1 \ 5 \ 2 \ 0$$

$$av = av - Request$$

$$= 1 \ 5 \ 2 \ 0 - 0 \ 4 \ 2 \ 0$$

$$= 1 \ 1 \ 0 \ 0$$

$$Allocation = All + Req$$

$$= 1 \ 0 \ 0 \ 0 + 0 \ 4 \ 2 \ 0$$

$$= 1 \ 4 \ 2 \ 0$$

$$Need = Need - Request$$

$$= 0 \ 7 \ 5 \ 0 - 0 \ 4 \ 2 \ 0$$

$$= 0 \ 3 \ 3 \ 0$$

3.1 \rightarrow New need Matrix
3.2 \rightarrow New Matrix

$$P_0 \Rightarrow 0 \ 0 \ 0 \ 0 \leq 1 \ 5 \ 2 \ 0 \rightarrow \text{True satisfied}$$

$$available = available + allocation$$

$$= 1 \ 5 \ 2 \ 0 + 0 \ 0 \ 1 \ 2$$

$$= 1 \ 5 \ 3 \ 2$$

$$P_1 \Rightarrow 0 \ 7 \ 5 \ 0 \leq 1 \ 5 \ 3 \ 2 \rightarrow \text{False N.S}$$

$$P_2 \Rightarrow 1 \ 0 \ 0 \ 2 \leq 1 \ 5 \ 3 \ 2 \rightarrow \text{True S.}$$

$$av = av + al$$

$$= 1 \ 5 \ 3 \ 2 + 1 \ 3 \ 5 \ 4 = 2 \ 8 \ 8 \ 6$$

$$P_3 \Rightarrow 0 \ 0 \ 2 \ 0 \leq 2 \ 8 \ 8 \ 6 \rightarrow \text{T. S.}$$

$$av = av + al$$

$$= 2 \ 8 \ 8 \ 6 + 0 \ 6 \ 5 \ 2 = 2 \ 14 \ 13 \ 8$$

$$P_4 \Rightarrow 0 \ 6 \ 4 \ 2 \leq 2 \ 14 \ 13 \ 8 \rightarrow \text{T. S}$$

$$av = av + al$$

$$= 2 \ 14 \ 13 \ 8 + 0 \ 6 \ 5 \ 6$$

$$= 2 \ 20 \ 18 \ 14$$

$$P_1 \Rightarrow 0 \ 7 \ 5 \ 0 \leq 2 \ 20 \ 18 \ 14 \rightarrow \text{T. S.}$$

$$av = av + al$$

$$= 2 \ 20 \ 18 \ 14 + 1 \ 7 \ 5 \ 0$$

$$= 3 \ 27 \ 23 \ 14$$