ECS 235B, Lecture 15

February 11, 2019
Constraint-Based Model (Yu-Gligor)

• Framed in terms of users accessing a server for some services
• *User agreement*: describes properties that users of servers must meet
• *Finite waiting time policy*: ensures no user is excluded from using resource
User Agreement

• Set of constraints designed to prevent denial of service
• $S_{seq}$ sequence of all possible invocations of a service
• $U_{seq}$ set of sequences of all possible invocations by a user
• $U_{li,seq} \subseteq U_{seq}$ that user $U_i$ can invoke
  • $C$ set of operations $U_i$ can perform to consume service
  • $P$ set of operations to produce service user $U_i$ consumes
  • $p < c$ means operation $p \in P$ must precede operation $c \in C$
  • $A_i$ set of operations allowed for user $U_i$
  • $R_i$ set of relations between every pair of allowed operations for $U_i$
Example

Mutually exclusive resource

- \( C = \{ \textit{acquire} \} \)
- \( P = \{ \textit{release} \} \)
- For \( p_1, p_2, A_i = \{ \textit{acquire}_i, \textit{release}_i \} \) for \( i = 1, 2 \)
- For \( p_1, p_2, R_i = \{ (\textit{acquire}_i < \textit{release}_i) \} \) for \( i = 1, 2 \)
Sequences of Operations

- $U_i(k)$ initial subsequence of $U_i$ of length $k$
  - $n_o(U_i(k))$ number of times operation $o$ occurs in $U_i(k)$
- $U_i(k)$ safe if the following 2 conditions hold:
  - if $o \in U_{i,seq}$, then $o \in A_i$; and
    - That is, if $U_i$ executes $o$, it must be an allowed operation for $U_i$
  - for all $k$, if $(o < o') \in R_i$, then $n_o(U_i(k)) \geq n_{o'}(U_i(k))$
    - That is, if one operation precedes another, the first one must occur more times than the second
Resources of Services

- $s \in S_{seq}$ possible sequence of invocations of services
- $s$ blocks on condition $c$
  - May be waiting for service to become available, or processing some response, etc.
- $o_i^*(c)$ represents operation $o_i$ blocked, waiting for $c$ to become true
  - When execution results, $o_i(c)$ represents operation
  - Note that when $c$ becomes true, $o_i^*(c)$ may not resume immediately
Resources of Services

• $s(0)$ initial subsequence of $s$ up to operation $o_i^*(c)$
• $s(k)$ subsequence of operations between $k$-1$^{st}$, $k^{th}$ time $c$ becomes true after $o_i^*(c)$
• $o_i^*(c) \rightarrow s(k) o_i(c)$: $o_i$ blocks waiting on $c$ at end of $s(0)$, resumes operation at end of $s(k)$
• $S_{seq\ live}$ if for every $o_i^*(c)$ there is a set of subsequences $s(0), ..., s(k)$ such that it is initial subsequence of some $s \in S_{seq}$ and $o_i^*(c) \rightarrow s(k) o_i(c)$
Example

• Mutually exclusive resource; consider sequence
  
  \((\text{acquire}_i, \text{release}_i, \text{acquire}_i, \text{acquire}_i, \text{release}_i)\)

  with \(\text{acquire}_i, \text{release}_i \in A_i, (\text{acquire}_i, \text{release}_i) \in R_i, o = \text{acquire}_i, o' = \text{release}_i\)

• \(U_i(1) = (\text{acquire}_i) \Rightarrow n_o(U_i(1)) = 1, n_{o'}(U_i(1)) = 0\)

• \(U_i(2) = (\text{acquire}_i, \text{release}_i) \Rightarrow n_o(U_i(2)) = 1, n_{o'}(U_i(2)) = 1\)

• \(U_i(3) = (\text{acquire}_i, \text{release}_i, \text{acquire}_i) \Rightarrow n_o(U_i(3)) = 2, n_{o'}(U_i(3)) = 1\)

• \(U_i(4) = (\text{acquire}_i, \text{release}_i, \text{acquire}_i, \text{acquire}_i) \Rightarrow\)
  
  \(n_o(U_i(4)) = 3, n_{o'}(U_i(4)) = 1\)

• \(U_i(5) = (\text{acquire}_i, \text{release}_i, \text{acquire}_i, \text{acquire}_i, \text{release}_i) \Rightarrow\)
  
  \(n_o(U_i(5)) = 3, n_{o'}(U_i(5)) = 2\)

• As \(n_o(U_i(k)) > n_{o'}(U_i(k))\) for \(k = 1, \ldots, 5\), the sequence is safe
Example (con’t)

• Let $c$ be true whenever resource can be released
  • That is, initially and whenever a $release_i$ operation is performed

• Consider sequence: $(acquire_1, acquire_2^*(c), release_1, release_2, \ldots, acquire_k, acquire_{k+1}(c), release_k, release_{k+1}, \ldots)$

• For all $k \geq 1$, $acquire_i^*(c) \xrightarrow{s(1)} acquire_{k+1}(c)$, so this is live sequence
  • Here, $acquire_{k+1}(c)$ occurs between $release_k$ and $release_{k+1}$
Expressing User Agreements

• Use temporal logics

• Symbols
  • $\square$: henceforth (the predicate is true and will remain true)
  • $\Diamond$: eventually (the predicate is either true now, or will become true in the future)
  • $\leadsto$: will lead to (if the first part is true, the second part will eventually become true); so $A \leadsto B$ is shorthand for $A \Rightarrow \Diamond B$
Example

• Acquiring and releasing mutually exclusive resource type

• User agreement: once a process is blocked on an \textit{acquire} operation, enough \textit{release} operations will release enough resources of that type to allow blocked process to proceed

\textbf{service} resource\_allocator

\textbf{User agreement}

\[ in(\textit{acquire}) \sim (\Box \Diamond (\#\textit{active\_release} > 0) \lor (\textit{free} \geq \textit{acquire.n})) \]

• When a process issues an \textit{acquire} request, at some later time at least 1 \textit{release} operation occurs, and enough resources will be freed for the requesting process to acquire the needed resources
Finite Waiting Time Policy

• *Fairness policy*: prevents starvation; ensures process using a resource will not block indefinitely if given the opportunity to progress

• *Simultaneity policy*: ensures progress; provides opportunities process needs to use resource

• *User agreement*: see earlier

• If these three hold, no process will wait an indefinite time before accessing and using the resource
Example

- Continuing example ... these and above user agreement ensure no indefinite blocking

**sharing policies**

**fairness**

\[(\text{at}(\text{acquire}) \land \lozenge\lozenge (\text{free} \geq \text{acquire}.n) \land (\#\text{active} = 0)) \Rightarrow \text{after}(\text{acquire})\]

\[(\text{at}(\text{release}) \land \lozenge\lozenge (\#\text{active} = 0)) \Rightarrow \text{after}(\text{release})\]

**simultaneity**

\[(\text{in}(\text{acquire}) \land (\lozenge\lozenge (\text{free} \geq \text{acquire}.n)) \land (\lozenge\lozenge (\#\text{active} = 0))) \Rightarrow \]

\[(((\text{free} \geq \text{acquire}.n) \land (\#\text{active} = 0))\]

\[(\text{in}(\text{release}) \land \lozenge\lozenge (\#\text{active}_{\text{release}} > 0)) \Rightarrow (\text{free} \geq \text{acquire}.n)\]
Service Specification

• Interface operations
• Private operations not available outside service
• Resource constraints
• Concurrency constraints
• Finite waiting time policy
Example:

- Interface operations of the resource allocation/deallocation example

**interface operations**

`acquire(n: units)`

**exception conditions:** `quota[id] < own[id] + n`

**effects:**

- `free' = free - n`
- `own[id]' = own[id] + n`

`release(n: units)`

**exception conditions:** `n > own[id]`

**effects:**

- `free' = free + n`
- `own[id]' = own[id] - n`
Example (con’t)

• Resource constrains of the resource allocation/deallocation example

resource constraints

1. $\square ((\text{free} \geq 0) \land (\text{free} \leq \text{size}))$

2. $(\forall id) \left[ \square (\text{own}[id] \geq 0) \land (\text{own}[id] \leq \text{quota}[id]) \right]$

3. $(\text{free} = N) \Rightarrow ((\text{free} = N) \text{ UNTIL } (\text{after} (\text{acquire}) \lor \text{after} (\text{release})))$

4. $(\forall id) \left[ (\text{own}[id] = M) \Rightarrow ((\text{own}[id] = M) \text{ UNTIL } (\text{after} (\text{acquire}) \lor \text{after} (\text{release}))) \right]$
Example (con’t)

• Concurrency constraints of the resource allocation/deallocation example

**Concurrency Constraints**

1. □(#active \leq 1)
2. (#active = 1) \sim (#active = 1)
Denial of Service

• Service specification policies, user agreements prevent denial of service *if enforced*

• These do *not* prevent a long wait time; they simply ensure the wait time is finite
State-Based Model (Millen)

• Unlike constraint-based model, allows a maximum waiting time to be specified
• Based on resource allocation system, denial of service base that enforces its policies
Resource Allocation System Model

- $R$ set of resource types
- For each $r \in R$, number of resource units (capacity, $c(r)$) is constant; a process can hold a unit for a maximum holding time $m(r)$
- $P$ set of processes
- For each $p \in P$, state is running or sleeping
  - When allocated a resource, process is running
  - Multiple process can be in running state simultaneously
  - Each $p$ has upper bound it can be in running state before being interrupted, if only by CPU quantum $q$
  - Example: if CPU considered a resource, $m(CPU) = q$
Allocation Matrix

• Rows represent processes; columns represent resources
  • $A: P \times R \rightarrow \mathbb{N}$ is matrix
  • For $p \in P$, $r \in R$, $A_p(r)$ is number of resource units of type $r$ acquired by $p$
  • As at most $c(r)$ of resource type $r$ exist, at most that many can be allocated at any time

R1: The system cannot allocate more instances of a resource type than it has:

$$(\forall r \in R)[\sum_{p \in P} A_p(r) \leq c(r)]$$
More About Resources

- **T**: \( P \rightarrow \mathbb{N} \) is system time when resource assignment was last changed
  - Think of it as a time vector, each element belonging to one process
- **Q^S**: \( P \times R \rightarrow \mathbb{N} \) is matrix of required resources for each process, *not including the resources it already holds*
  - So \( Q^S_p(r) \) means the number of units of resource type \( r \) that process \( p \) may need to complete
- **Q^T**: \( P \times R \rightarrow \mathbb{N} \) is matrix of how much longer each process \( p \) needs the units of resource \( r \)
- Predicates \( \text{running}(p) \) true if \( p \) is in running state; \( \text{asleep}(p) \) true otherwise

R2: A currently running process must not require additional resources to run

\[
\text{running}(p) \Rightarrow (\forall r \in R)[Q^S_p(r) = 0]
\]
States, State Transitions

• Current state of system is \((A, T, Q^S, Q^T)\)

• State transition \((A, T, Q^S, Q^T) \rightarrow (A', T', Q^{S'}, Q^{T'})\)
  • We only care about transitions due to allocation, deallocation of resources

• Three relevant types of transitions
  • Deactivation transition: \(\text{running}(p) \rightarrow \text{asleep'}(p)\); process stops execution
  • Activation transition: \(\text{asleep}(p) \rightarrow \text{running'}(p)\); process starts or resumes execution
  • Reallocation transition: transition in which \(p\) has resource allocation changed; can only occur when \(\text{asleep}(p)\)
Constraints

R3: Resource allocation does not affect allocations of a running process:

\[(\text{running}(p) \land \text{running}'(p)) \Rightarrow (A_p' = A_p)\]

R4: \(T(p)\) changes only when resource allocation of \(p\) changes:

\[(A_p'(CPU) = A_p(CPU)) \Rightarrow (T'(p) = T(p))\]

R5: Updates in time vector increase value of element being updated:

\[(A_p'(CPU) \neq A_p(CPU)) \Rightarrow (T'(p) > T(p))\]
Constraints

R6: When $p$ reallocated resources, allocation matrix updated before $p$ resumes execution:

$$\text{asleep}(p) \Rightarrow Q_{p}' = Q_{p} + A_p - A'_p$$

R7: When a process is not running, the time it needs resources does not change:

$$\text{asleep}(p) \Rightarrow Q_{p}' = Q_{p}$$

R8: when a process ceases to execute, the only resource it must surrender is the CPU:

$$(\text{running}(p) \land \text{asleep}'(p)) \Rightarrow A_p'(r) = A_p(r) - 1 \quad \text{if } r = \text{CPU}$$

$$(\text{running}(p) \land \text{asleep}'(p)) \Rightarrow A_p'(r) = A_p(r) \quad \text{otherwise}$$
Resource Allocation System

• A system in a state \((A, T, Q^S, Q^T)\) such that:
  • State satisfies constraints R1, R2
  • All state transitions constrained to meet R3-R8
Denial of Service Protection Base (DPB)

• A mechanism that is tamperproof, cannot be prevented from operating, and guarantees authorized access to resources it controls

• Four parts:
  • Resource allocation system (see earlier)
  • Resource monitor
  • Waiting time policy
  • User agreement (see earlier; constraints apply to changes in allocation when process transitions from $running(p)$ to $asleep(p)$
Resource Monitor

- Controls allocation, deallocation of resources and the timing
- $Q^S_p$ is feasible if $(\forall i)[Q^S_p(r_i) + A_p(r_i) \leq c(r_i)] \land Q^S_p(CPU) \leq 1$
  - If the total number of resources it will be allocated will always be no more than the capacity of that resource, and no more than 1 CPU is requested
- $T_p$ is feasible if $(\forall i)[T_p(r_i) \leq \text{max}(r_i)]$
  - Here, $\text{max}(r_i)$ max time a process must wait for its needed allocation of units of resource type $i$
Waiting Time Policy

• Let $\sigma = (A, T, Q_S, Q_T)$

• Example finite waiting time policy:
  
  $\forall p, \sigma (\exists \sigma') [running'(p) \land (T'(p) \geq T(p))]$
  
  • For every process and state, there is a future state in which $p$ is executing and has been allocated resources

• Example maximum waiting time policy:
  
  $\exists M (\forall p, \sigma (\exists \sigma') [running'(p) \land (0 < T'(p) - T(p) \leq M)])$
  
  • There is an upper bound $M$ to how long it takes every process to reach a future state in which it is executing and has been allocated resources
Two Additional Constraints

In addition to all these, a DPB must satisfy these constraints:

1. Each process satisfying user agreement constraints will progress in a way that satisfies the waiting time policy

2. No resource other than the CPU is deallocated from a process unless that resource is no longer needed

\[(\forall i) [r_i \neq \text{CPU} \land A_p(r_i) \neq 0 \land A_p'(r_i) = 0] \Rightarrow Q^T_p(r_i) = 0\]
Example: DPB

- Assume system has 1 CPU
- Assume maximum waiting time policy in place
- 3 parts to user agreement:
  - $Q^S_p, T_p$ are feasible
  - Process in running state executes for a minimum amount of time before it transitions to a non-running state
  - If process requires resource type, and enters a non-running state, the time it needs the resource for is decreased by the amount of time it was in the previous running state; that is,

$$Q^T_p \neq 0 \land running(p) \land asleep'(p) \Rightarrow (\forall r \in R)[Q^T_p(r) \leq \max(0, \max_r Q^T_p(r) - (T'(p) - T(p)))]$$
Example: System

- $n$ processes, round robin scheduler with quantum $q$
- Initially no process has any resources
- Resource monitor selects process $p$ to give resources to
  - $p$ executes until $Q_p^T = 0$ or monitor concludes $Q_p^S$ or $T_p$ is not feasible
- Goal: show there will be no denial of service in this system because
  a) no resource $r_i$ is deallocated from $p$ for which $Q_p^S$ is feasible until $Q_p^T = 0$; and
  b) there is a maximum time for each round robin cycle
Claim (a)

• Before $p$ selected, no process has any resources allocated to it
  • So next process with $Q^S_p$ and $T_p$ feasible is selected
  • It runs until it enters the *asleep* state or $q$, whichever is shorter
  • If in *asleep* state, process is done
  • If $q$, monitor gives $p$ another quantum of running time; this repeats until $Q^T_p = 0$, and then $p$ needs no more resources

• Let $m(r)$ be maximum time any process will hold resources of type $r$
  • Let $M(r) = \max_r m(r)$

• As $Q^S_p$ and $T_p$ feasible, $M$ upper bound for all elements of $Q^T_p$
  • $d = \min(q)$, minimum time before $p$ transitions to *asleep* state); exists because a process in running state executes for a minimum amount of time before it transitions to a non-running state
Claim (a) (con’t)

• As $Q^S_p$ and $T_p$ feasible, $M$ upper bound for all elements of $Q^T_p$
• $d = \min(q, \text{minimum time before } p \text{ transitions to asleep state})$
  • Exists because a process in running state executes for a minimum amount of
    time before it transitions to a non-running state
• At end of each quantum, $m'(r) = m(r) - d$
  • By third part of user agreement
• So after $\text{floor}(M/d + 1)$ quanta, $Q^T_p = 0$
  • So no resources deallocated until $(\forall i) \ Q^T_p(r_i) = 0$
Claim (b)

• $t_a$ is time between resource monitor beginning cycle and when it has allocated required resources to $p$

• Resource monitor then allocates CPU resource to $p$; call this time $t_{CPU}$
  • Done between each quantum

• When $p$ completes, all its resources deallocated; this takes time $t_d$

• As $Q_s^p$ and $T_p$ feasible, time needed to run $p$, including time to deallocate all resources, is:

$$t_a + \text{floor}(M/d + 1)(q + t_{CPU}) + t_d$$

• So for $n$ processes, maximum time cycle will take is $n$ times this

• Thus, there is a maximum time for each round robin cycle
Availability and Network Flooding

• Access over Internet must be unimpeded
  • Context: flooding attacks, in which attackers try to overwhelm system resources
• If many sources flood a target, it’s a *distributed denial of service attack*
TCP 3-Way Handshake and Availability

- Normal three-way handshake to initiate connection
- Suppose source never sends third message (the last ACK)
  - Destination holds information about pending connection for a period of time before the space is released
Analysis

• Consumption of bandwidth
  • If flooding overwhelms capacity of physical network medium, SYNs from legitimate handshake attempts may not be able to reach the target

• Absorption of resources on destination host
  • Flooding fills up memory space for pending connections, causing SYNs from legitimate handshake attempts to be discarded

• In terms of the models:
  • Waiting time is the time that destination waits for ACK from source
  • Fairness policy must assure host waiting for ACK (resource) will receive (acquire) it
Analysis in Terms of Model

• Waiting time is the time that destination waits for ACK from source
• Fairness policy must assure host waiting for ACK (resource) will receive (acquire) it
  • But goal of attack is to make sure it never arrives
• Yu-Gligor model: finite wait time does not hold
  • So model says denial of service can occur
• Millen model: $T_p(ACK) > max(ACK)$
  • $max(ACK)$ is the time-out period for pending connections
  • So model says denial of service can occur
Countermeasures

- Focus on ensuring resources needed for legitimate handshakes to complete are available
  - So every legitimate client gets access to server
- First approach: manipulate opening of connection at end point
  - If focus is to ensure connection attempts will succeed at some time, focus is really on waiting time
  - Otherwise, focus is on user agreement
- Second approach: control which packets, or rate at which packets, sent to destination
  - Focus is on implicit user agreements
Intermediate Systems

• Approach is to reduce consumption of resources on destination by diverting or eliminating illegitimate traffic so only legitimate traffic reaches destination
  • Done at infrastructure level

• Example: Cisco routers try to establish connection with source (TCP intercept mode)
  • On success, router does same with intended destination, merges the two
  • On failure, short time-out protects router resources and target never sees flood
Track Connection Status

• Use network monitor to track status of handshake

• Example: synkill monitors traffic on network
  • Classifies IP addresses as not flooding (good), flooding (bad), unknown (new)
  • Checks IP address of SYN
    • If good, packet ignored
    • If bad, send RST to destination; ends handshake, releasing resources
    • If new, look for ACK or RST from same source; if seen, change to good; if not seen, change to bad

• Periodically discard stale good addresses
Intermediate Systems near Sources

- D-WARD relies on routers close to the sources to block attack
  - Reduces congestion in network without interfering with legitimate traffic
- Placed at gateways of possible sources to examine packets leaving (internal) network and going to Internet
- Deployed on systems in research lab for 4 months
  - First month: large number of false alerts
  - Tuning D-WARD parameters reduced this number
D-WARD: Observation Component

• Has set of legitimate internal addresses
• Gathers statistics on packets leaving network, discarding packets without legitimate addresses
• Tracks number of simultaneous connections to each remote destination
  • Unusually large number may indicate attack from this network
• Examines connections with large amount of outgoing traffic but little incoming (response) traffic
  • May indicate destination host is overwhelmed
D-WARD: Observation Component

- Also aggregates traffic statistics to each remote address
- Classifies flows as *attack, suspicious, normal*
  - *Normal*: statistics match legitimate traffic model
  - *Attack*: if not
- Once traffic classified as attack begins to match legitimate traffic model, indicates attack has ended, so flow reclassified as *suspicious*
  - If it stays suspicious for predetermined time, reclassified as *normal*
D-WARD: Rate-Limiting Component

• When attack detected, this component limits amount of packets that can be sent

• This reduces volume of traffic going from this network to destination

• How it limits rate is based on D-WARD’s best guess of amount of traffic destination can handle
  • When flow reclassified as normal, D-WARD raises rate limit until sending rate is as before
D-WARD: Traffic-Policing Component

- Component obtains information from other 2 components
- Based on this, decides whether to drop packets
  - Packets for normal connections always forwarded
  - Packets for other flows may be forwarded provided doing so does not exceed rate limit associated with flow