# CSCI 4730/6730 OS
## (Chap #8 Deadlocks)

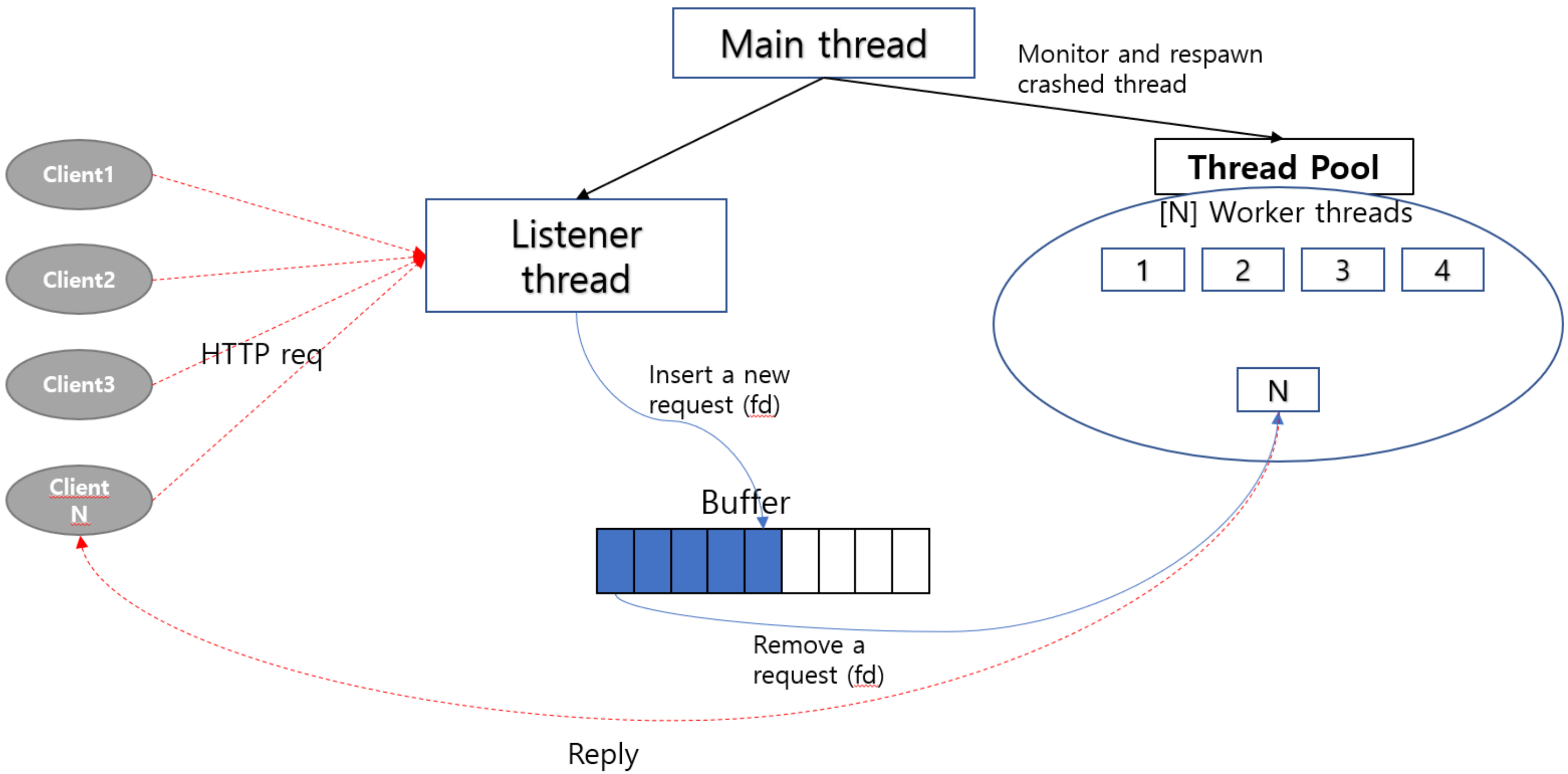
In Kee Kim

Department of Computer Science

University of Georgia

# PA #2

- Multi-threaded Webserver with Sync



# Requirements

- Multi-Threading
  - Thread Pool
- Producer-Consumer Model
  - Two semaphores; **`sem_empty, sem_full`**
  - One mutex lock; **`mutex`**

# Consumer-Producer Model

- **`./webserver_multi 5001 10`**
- You need to create **<u>1</u>** producer and **<u>10</u>** consumers
  - 1 Producer: part of listener
    - Insert “http request” to buffer
  - N Consumer: worker thread
    - Read “http request” from buffer
    - Call process() (in net.c)

# webserver.c

```c
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <sys/socket.h>
#include <linux/unistd.h>
#include <arpa/inet.h>
#include "webserver.h"

int listener(int port)
{
  int sock;
  struct sockaddr_in sin;
  struct sockaddr_in peer;



  sock = socket(AF_INET, SOCK_STREAM, 0);
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = INADDR_ANY;
  sin.sin_port = htons(port);
  r = bind(sock, (struct sockaddr *) &sin, sizeof(sin));
  if(r < 0) {
    perror("Error binding socket:");
    return -1;
  }

  r = listen(sock, 5);
  if(r < 0) {
    perror("Error listening socket:");
    return -1;
  }

  printf("HTTP server listening on port %d\n", port);
  while (1) {
    int fd;
    fd = accept(sock, NULL, NULL);
    if (fd < 0) {
      printf("Accept failed.\n");
      break;
    }
    process(fd);
  }
  close(sock);

}

int main(int argc, char *argv[]) {
  if(argc != 2 || atoi(argv[1]) < 2000 || atoi(argv[1]) > 50000)
  {
    fprintf(stderr, "./webserver PORT(2001 ~ 49999)\n");
    return 0;
  }

  int port = atoi(argv[1]);
  listener(port);
  return 0;
}
```

# webserver_multi.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <pthread.h>
#include "webserver.h"

#define MAX_REQUEST 100

int port, numThread;

void *listener()
{
  int r;
  struct sockaddr_in sin;
  struct sockaddr_in peer;
  int peer_len = sizeof(peer);
  int sock;

  sock = socket(AF_INET, SOCK_STREAM, 0);
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = INADDR_ANY;
  sin.sin_port = htons(port);
  r = bind(sock, (struct sockaddr *) &sin, sizeof(sin));
  if(r < 0) {
    perror("Error binding socket:");
    return;
  }

  r = listen(sock, 5);
  if(r < 0) {
    perror("Error listening socket:");
    return;
  }

  printf("HTTP server listening on port %d\n", port);
  while (1)
  {
    int s;
    s = accept(sock, NULL, NULL);
    if (s < 0) break;

    //process(s); producer(s)
  }
  close(sock);
}

void thread_control()
{
  /* ----- */
}

int main(int argc, char *argv[])
{
  if(argc != 3 || atoi(argv[1]) < 2000 || atoi(argv[1]) > 50000)
  {
    fprintf(stderr, "./webserver_multi PORT(2001 ~ 49999) #_of_threads\n");
    return 0;
  }

  int i;
  port = atoi(argv[1]);
  numThread = atoi(argv[2]);
  thread_control();
  return 0;
}
```

# First thing to do

- Create Buffer with Size of **`MAX_REQUEST`** (100)
  - This is basically an array.
- What should be stored in buffer?
  - http request?
  - What is data (var) type of http request?
    - What would be data (var) type for buffer?
  - Everything you need is in webserver.c
  - **`man 2 listen, man 2 accept`**

# `thread_control()`

- Create threads
  - 1 call of “pthread_create” for listener
  - N calls of “pthread_create” for workers (consumer)
- Pthread Join

# Producer(s)

- Caller is “listener”
- What is variable type for “s”?
- Scan buffer, and insert “s” if there is empty slot
  - Ok to use linear search
- Semaphore and Mutex!!!
  - Blocking if buffer is full

# `Consumer()`

- Worker thread
- Blocking if buffer is empty
  - Semaphore!
- Scan buffer, if “http request” is in the buffer
  - Read the request
  - Call process()!
  - Same – linear search is ok
  - After read a req from buffer, please remove it

# Critical Section

- Buffer management
  - Insert **req** to buffer
  - Read and remove **req** from buffer

# Chapter 8: Deadlock

- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

# Chapter 8: Deadlock

- Note
  - The term "**process**" and "**thread**" can be used *interchangeably*.

# What is Deadlock?

- A condition where two or more threads (processes) are waiting for an event that can only be generated by these same threads.

```
P0                    P1
wait (S);             wait (Q);
wait (Q);             wait (S);

...                   ...

signal(S);            signal(Q)
signal(Q);            signal(S)
```

# Deadlock

- What is the result of Deadlock?
- Can CPU scheduler address Deadlock? In other words, CPU scheduler makes progress of programs in deadlock condition?

# Deadlocks: Terminology

- **Deadlock Detection**
  - Finding instances of deadlock when threads stop making process and tries to recover
- **Deadlock Prevention**
  - Imposing restrictions on program to prevent the possibility of deadlock
- **Deadlock Avoidance**
  - Using algorithms that check resource requests and availability at runtime to avoid deadlock

# Deadlocks: Terminology

- **Starvation**
  - Occurs when a thread waits indefinitely for some resources, but other threads are actually using the resources
- **Starvation == Deadlock?**
- In starvation, program makes progress or not?
- In deadlock, program makes progress or not?

# Necessary Conditions for Deadlock

- Prerequisite
  - Multiple Threads/Processes
  - Finite resources
  - Multiple Threads/Procs are competing for resources
- Deadlock can happen if **all** the following conditions hold
  - **Mutual Exclusion**
  - **Hold and Wait**
  - **No Preemption**
  - **Circular wait**

# Necessary Conditions for Deadlock

- Deadlock can happen if **all** the following conditions hold
  - **Mutual Exclusion:** at least one thread must hold a resource in non-sharable mode.
    - i.e., the resource may only be used by one thread at a time
  - **Hold and Wait:** at least one thread holds a resource and is waiting for other resource(s) to become available.
  - **No Preemption:** a thread can only release a resource voluntarily; another thread or the OS cannot force the thread to release the resource
  - **Circular wait:** a set of waiting threads $\{T_1,\ldots,T_n\}$ where $T_i$ is waiting on $T_{i+1}$ ($i$ = 1 to $n$) and $T_n$ is waiting on $T_1$.

# Deadlock Handling in OS

- Deadlock Detection and Recovery
- Deadlock Prevention
- Deadlock Avoidance

Expensive!!!

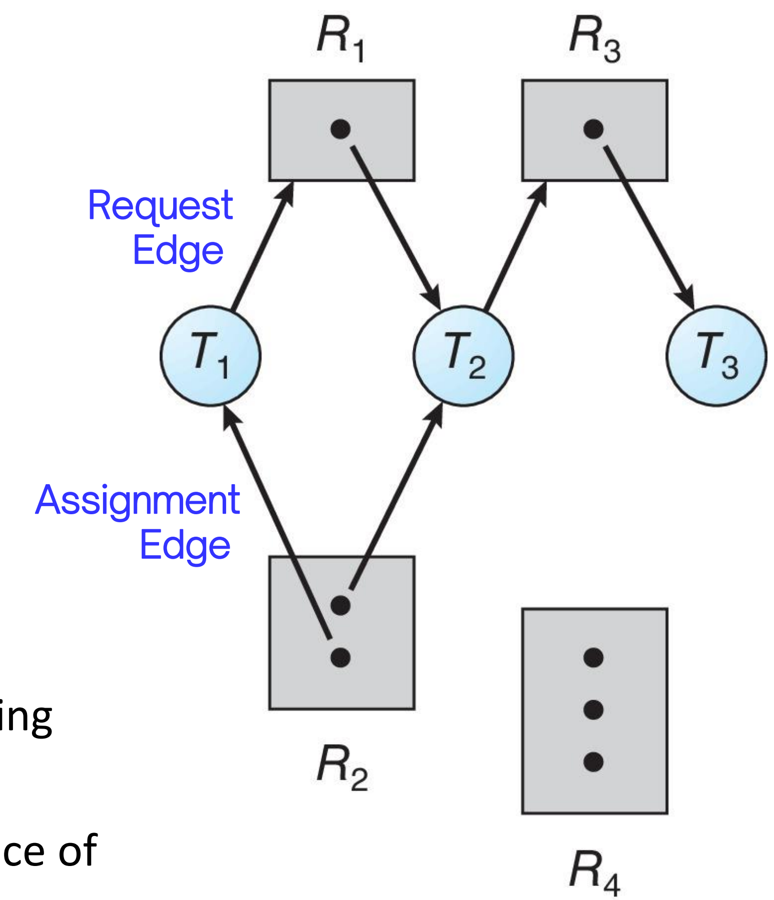
- Deadlock Ignorance

# Resource-Allocation Graph

- Can be used for deadlock detection and avoidance
- A graph composed of a set of vertices *V* and a set of edges *E*
  - Two types of Vertices
    - *Thread Set* $T = \{T_1, T_2, \ldots, T_n\}$,
    - *Resource Set* $R = \{R_1, R_2, \ldots, R_m\}$
- **Request Edge** – directed edge from thread $\boldsymbol{T_i}$ to resource $\boldsymbol{R_j}$
  - $\boldsymbol{T_i}$ has requested that resource, but has not yet required it
- **Assignment Edge** – directed edge from resource $\boldsymbol{R_j}$ to thread $\boldsymbol{T_i}$
  - OS has allocated $\boldsymbol{R_j}$ to $\boldsymbol{T_i}$

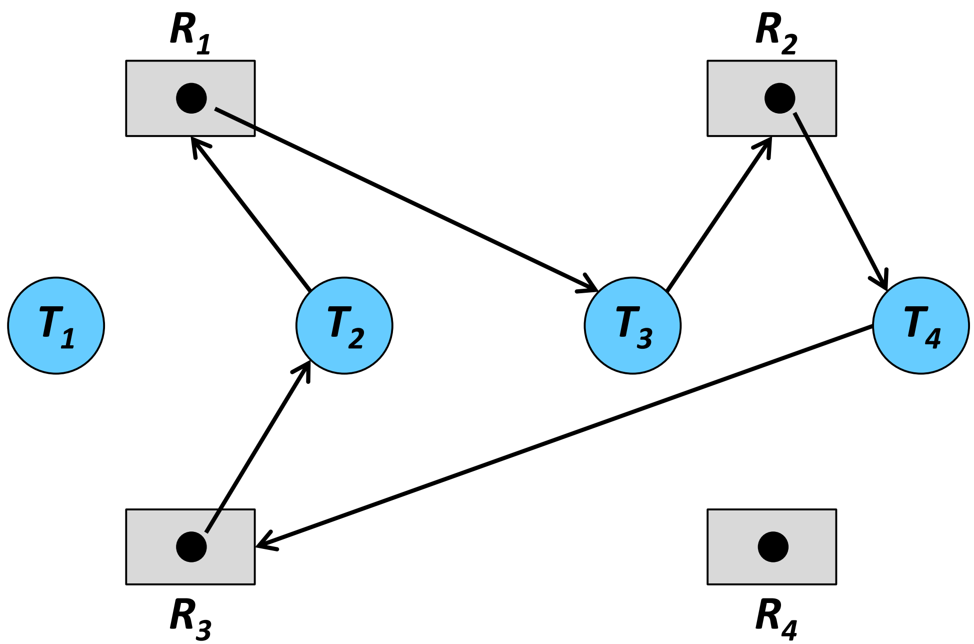
# Resource-Allocation Graph Example

- Resource instances:
  - The "●"
  - One instance of R1
  - Two instances of R2
  - One instance of R3
  - Three instance of R4
- Thread States:
  - T1 holds one instance of R2 and is waiting for an instance of R1
  - T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
  - T3 is holds one instance of R3



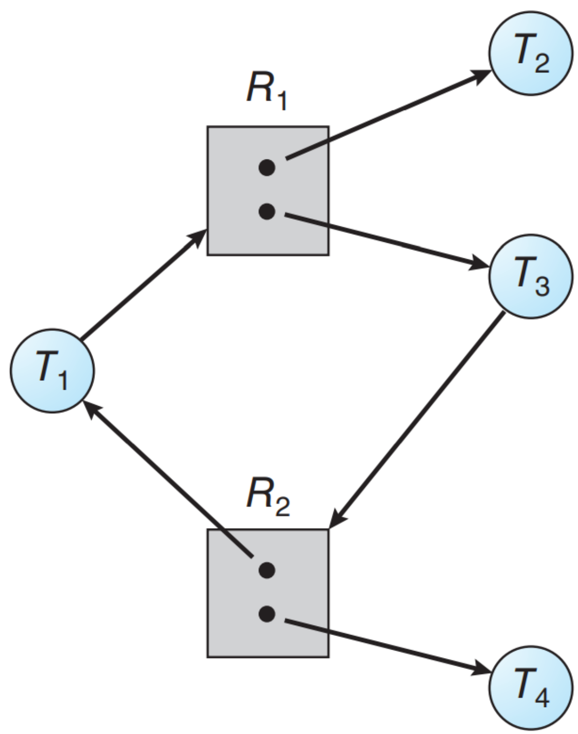
# Deadlock Detection Using Resource-Allocation Graph

- If the graph has no cycles, no deadlock exists
- If the graph has a cycle, deadlock ***might*** exist
- Wait-for Graph



# Deadlock Detection Using Resource-Allocation Graph

- If the graph has no cycles, no deadlock exists
- If the graph has a cycle, deadlock ***might*** exist



# Deadlock Detection Using Resource-Allocation Graph

- If the graph has no cycles, no deadlock exists
- If the graph has a cycle, deadlock ***might*** exist
- If **any instance of a resource involved in the cycle is held by a thread *not in the cycle***, then we can make progress when that resource is released
- **Wait-for Graph** – A Variant.

# Deadlock Prevention

- **Imposing restrictions on program to prevent the possibility of deadlock**
- Idea: at least one of four conditions cannot hold!
- Eliminate one (or more) of:
  - mutual exclusion
  - hold and wait
  - no preemption  (i.e., *have* preemption)
  - circular wait

# Deadlock Prevention

- Eliminate Mutual Exclusion
  - Shared resources do not require mutual exclusion
    - *e.g.,* read-only files
  - Potential Issue?
    - Some (most) resources **cannot be shared** (at the same time)

# Deadlock Prevention

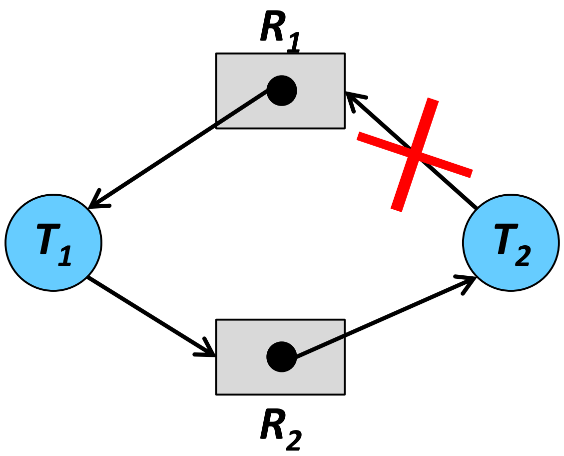
- Eliminate Hold & Wait – “Not Wait” or “Not Hold”
  - **Can you make “not-wait” policy?**
    - Each process holds all of its resources before it begins executing
    - Eliminates the **wait** possibility
    - Dining-Philosopher Problem
    - **`lock(r1,r2,…);`**
  - **Can you make “not-hold” policy?**
    - Alternatively, only allow a process to request resources when it currently has none
    - Eliminates the hold possibility
  - Problems??
    - Low Utilization or Starvation

# Deadlock Prevention

- Eliminate “No Preemption”
  - Similar to not-hold policy
  - Make a process automatically release its current resources if it cannot obtain all the ones it wants
    - Restart the process when it can obtain everything
  - **Alternatively, the desired resources can be preempted from other waiting processes**
  - Problem??
    - **Not all resources can be easily preempted, like printer, scanner..**

# Deadlock Prevention

- Eliminate Circular Wait
  - Impose an ordering (numbering) on the resources and request them in order
  - If a thread holds resource *i*, then it can always request and acquire resources > *i*
  - If a thread holds resource *i*, then it cannot request and acquire resources < *i*



# Problems in Deadlock Prevision

- EACH of these prevention techniques **may cause**
  - Low Resource Utilization
  - Starvation
  - Not implementable

- Not desired option

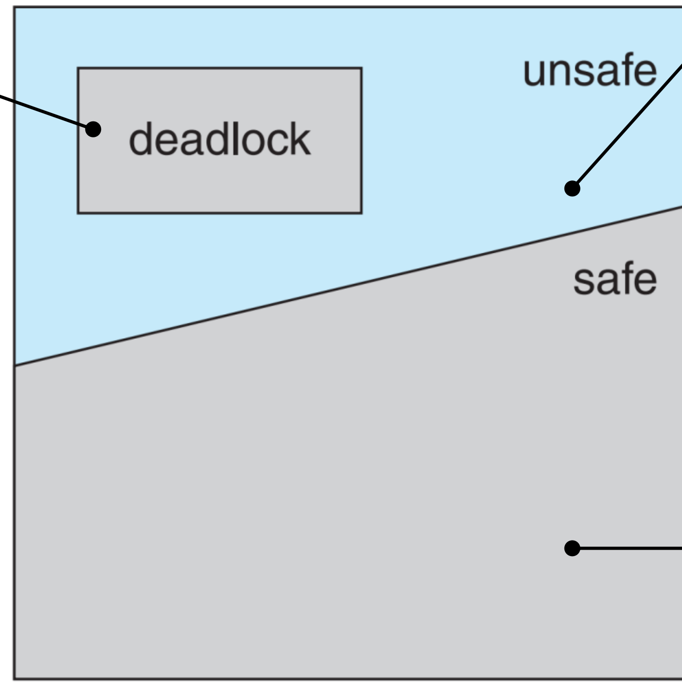
# Deadlock Avoidance

- Condition
  - If we or OS have prior knowledge of how resources will be requested, it is possible to determine if threads are entering an "unsafe" state.
- Possible states are:
  - **Deadlock**: no forward progress can be made
  - **Unsafe state**: a state that ***may*** allow deadlock
  - **Safe state**: a state is safe if there exists a **safe sequence**

# Deadlock Avoidance

May be!
No deadlock if lucky
Deadlock; if unlucky

Deadlock; No progress




No Deadlock;
Thread/Procs are safe

**NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks**

# Deadlock Avoidance

- **Rule is simple:** Ensure that a system will never enter an unsafe state.

- How? – OS analyzes resource allocation state to determine whether granting a request can lead to deadlock in future.
  - if not lead to deadlock, then granted
  - Otherwise, keep pending until they can be granted (**process/thread may face long delay for obtaining a resource**)

# Avoidance Algorithms

- Single instance of a resource type
  - e.g., a file, a printer, a scanner…
  - Use **resource-allocation graph**
- Multiple instances of a resource type
  - e.g., CPU, IO, MEM, Network BW…
  - Use **banker's algorithm**

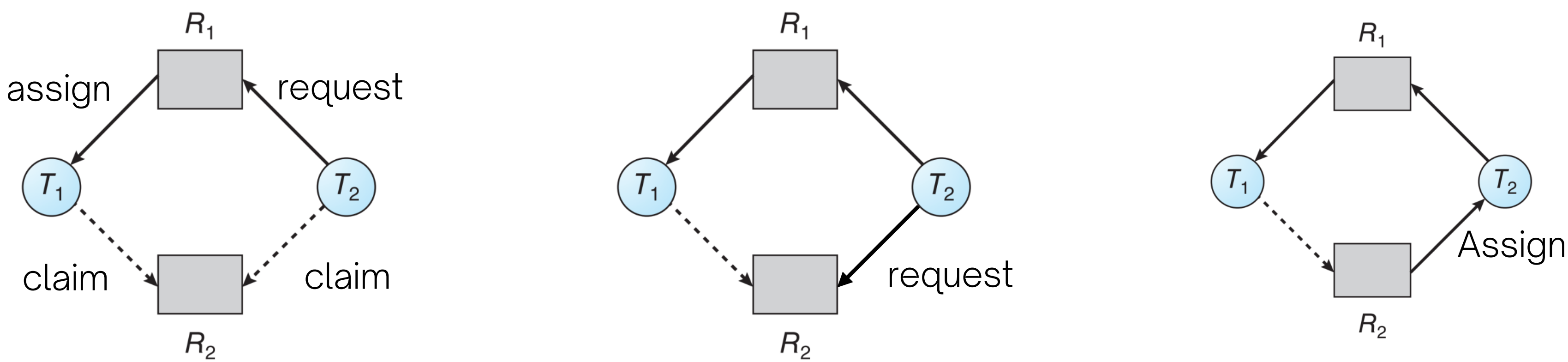
# Resource-Allocation Graph Variation

- Can be used for deadlock detection and avoidance
- A graph composed of a set of vertices *V* and a set of edges *E*
  - Two types of Vertices, $T = \{T_1, T_2, \ldots, T_n\}$, $R = \{R_1, R_2, \ldots, R_m\}$
- **Request Edge** – directed edge (solid line) from thread $\boldsymbol{T_i}$ to resource $\boldsymbol{R_j}$
  - $\boldsymbol{T_i}$ has requested that resource, but has not yet required it
  - $\boldsymbol{T_i \longrightarrow R_j}$
- **Assignment Edge** – directed edge (solid line) from resource $\boldsymbol{R_j}$ to thread $\boldsymbol{T_i}$
  - OS has allocated $\boldsymbol{R_j}$ to $\boldsymbol{T_i}$
  - $\boldsymbol{T_i \longleftarrow R_j}$
- **Claim Edge** – directed edge (dashed line) from thread $\boldsymbol{T_i}$ to resource $\boldsymbol{R_j}$
  - $\boldsymbol{T_i}$ ***may*** request that resource in the future, but not right now
  - $\boldsymbol{T_i}$ ----> $\boldsymbol{R_j}$

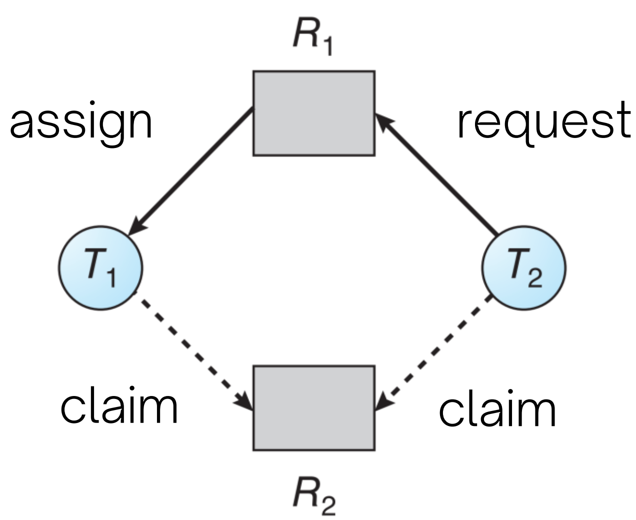
# Resource-Allocation Graph Variation

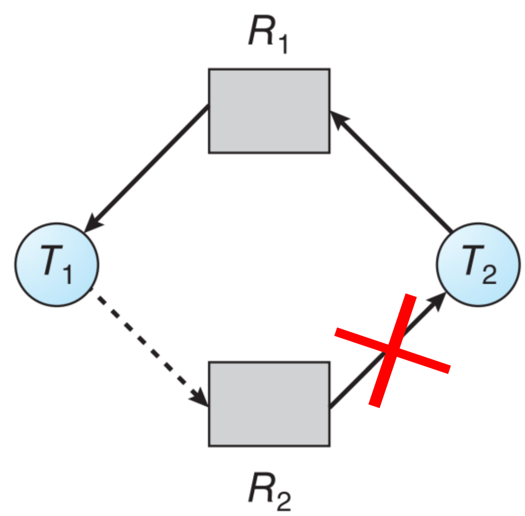
- **Claim edge** $T_i \dashrightarrow R_j$ indicated that thread $T_i$ may request resource $R_j$; represented by a dashed line
- Claim edge ($T_i \dashrightarrow R_j$) converts to request edge ($T_i \longrightarrow R_j$) when the thread $T_i$ requests the resource $R_j$
- Request edge ($T_i \longrightarrow R_j$) converted to an assignment edge ($T_i \longleftarrow R_j$) when the resource $R_j$ is allocated to the thread $T_i$
- Resources must be claimed ***a priori*** in the system

# Resource-Allocation Graph Variation



# Avoidance with Resource-Allocation Graph




- Suppose that thread $T_i$ requests a resource $R_j$
- The request can be granted only if converting the request edge to an assignment edge does ***not result in the formation of a cycle*** in the resource allocation graph

# Banker's Algorithm

- Use Banker's algorithm when multiple instances exist in a resource type
- Assume that:
  - A resource can have multiple instances
    - Pretty common in real systems. e.g., CPU, memory, disk, network
  - OS has ***N*** threads, ***M*** resource types
- Initially, each thread must declare the maximum number of resources it will need. Ugh -- ☹
- Calculate a safe sequence if possible.

# Banker's Algorithm

- Let's assume a very simple model:
  - Each thread declares its maximum needs.
  - If the solution of the algorithms exists, this ensures no unsafe state is reached.
- Note that maximum needs does *NOT* mean it must use that many resources – simply it might do so under some circumstances.

# Banker's Algorithm

- Simple Example
  - A system with **12** CPU resources. Each resource is used exclusively by a thread. The current state looks like this:

| Thread | Max Needs | Allocated | Current Needs |
|---|---|---|---|
| T0 | 10 | 5 | **?** |
| T1 | 4 | 2 | **?** |
| T2 | 7 | 3 | **?** |

$$\sum \boldsymbol{Allocated} = \mathbf{10}$$

In this example,
What is workable sequence?

**<T1, T2, T0>**

Suppose T2 requests and is given one more resource. What happens then?