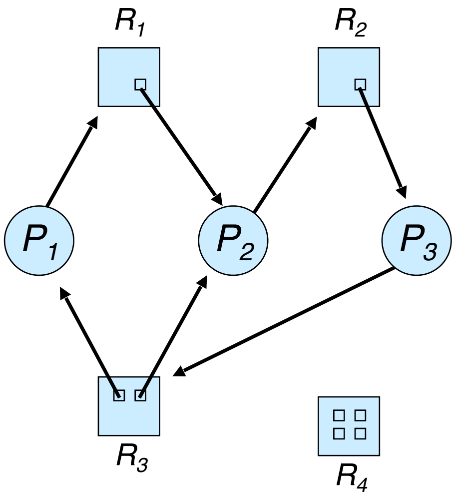
Concurrency

- Exam 1
- 26 - Concurrency
- 27 - Overview, and POSIX threads (pthreads)
- 28 - Locks
- 29 - Concurrent Data Structures
- 30 Condition Variables
- 31 - Semaphores
- 32 - Common Problems (including deadlocks)
- 33 - Event-Based Concurrency

Resource Allocation Graph *deadlock*



1. deadlock requires cycle
2. not all cycles are deadlocks (if multiple instances of some resources)

- $P_2 \rightarrow R_2 \rightarrow P_3 \rightarrow R_3 \rightarrow P_2$ cycle (and part of a deadlock)
- $R_3 \rightarrow P_1 \rightarrow R_1 \rightarrow P_2$ not a cycle

Handling Deadlocks *how to fix*

- What to do?
  - **prevent** by *constraining how* resource requests made
  - **avoid** by filtering dangerous actions *per-request*
  - **deal with** when they occur
  - **pretend** they never happen

Deadlock Prevention

- Try to prevent one of the four conditions from holding true
  - Difficult to eliminate mutual exclusion
  - Prevent threads from requesting new resources when holding other resources (eliminates hold and wait)
  - Require threads not immediately able to get all needed resources to give up those they have (eliminates no preemption)
  - Require agreed-upon resource acquisition ordering (eliminates circular waiting).

Prevents at least one of the conditions from holding by *constraining how* resource requests made.

Deadlock Prevention *hold and wait*

- Acquire all locks at once:

```
pthread_mutex_lock(prevention);          // begin acquisition
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention);        // end
```

- So we never wait, but?
  - `prevention` lock is global
  - need complete information on locks to be acquired

Deadlock Prevention *no preemption*

- *try* locks:
  - atomically grab lock if available, or return w/ error

```
top:
    pthread_mutex_lock(L1);                // begin acquisition
    if (pthread_mutex_trylock(L2) != 0) {
        pthread_mutex_unlock(L1);
        goto top;
    }
```

- Works even if other thread chooses different order.
- However: *livelock:*
  - Possible, though unlikely, that the threads both back off forever. Fix with random delays.
- Also *encapsulation*:
  - locks acquisitions may be hidden by function calls, making reset to initial state difficult
  - language approaches can work, or just avoid encapsulation

Deadlock Prevention *circular wait*

- Don’t do this:

```
T1: pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
```
①

```
T2: pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
```
②

- Agree on lexicographic ordering on lock acquisitions:

```
T2: pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
```

- or address-based:

```
if (m1 > m2) {        // grab in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
```

Deadlock Prevention *mutual exclusion*

- *Lock-free* and *wait-free* data structures and algorithms
  - use atomic instructions such as *CompareAndSwap*

// pseudocode of atomic assembly instruction

```
int CompareAndSwap(int *address, int expected, int new) {
    if (*address == expected) {
        *address = new;
        return 1;                         // success
    }
    return 0;                             // failure
}
```

- Atomically increment a counter w/o locks:

```
void AtomicIncrement(int *value, int amount) {
    do {
        int old = *value;
    } while (CompareAndSwap(value, old, old + amount) == 0);
}
```

Deadlock Prevention *more wait-free*

// mutex-based

```
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    n->value = value;
    pthread_mutex_lock(listlock);   // begin critical section
    n->next  = head;
    head     = n;
    pthread_mutex_unlock(listlock); // end critical section
}
```

// wait-free

```
void insert(int value) {
    node_t *n = malloc(sizeof(node_t)); assert(n != NULL);
    n->value = value;
    do{
        n->next = head;
    } while (CompareAndSwap(&head, n->next, n) == 0);
}
```

Deadlock Avoidance *safe states*

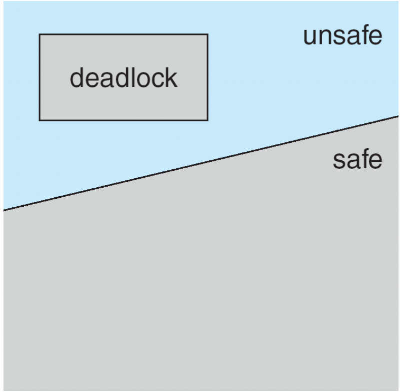
OS uses info on which resource requests a process might make to filter dangerous actions on a *per-request* basis.

- System is in *safe state* if there exists:
  - *safe sequence* $<P_1, P_2, \ldots, P_n>$ of ALL processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources *+ resources held by all P_j s.t. $j < i$*

- That is:
  - If P_i's resource needs are not immediately available, then P_i can wait until all P_j s.t. $j < i$ have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, …

Deadlock Avoidance *safe states*

- In other words:
  - *unsafe state* ⟶ converting a single request to a claim *can* result in deadlock
  - *safe state* ⟶ converting a single request *cannot* result in deadlock
- Avoidance of unsafe states ensures no deadlocks.



Deadlock Avoidance *safe states*

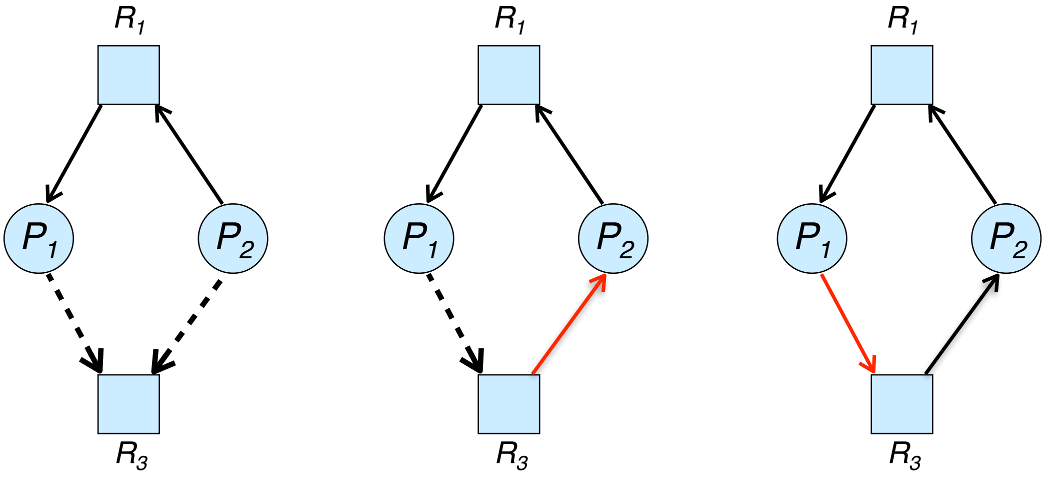
- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of resource types
  - Use Dijkstra's *banker algorithm*

Deadlock Avoidance *safe states*

- New claim edge $P_i \rightarrow R_j$ indicates P_i may request resource R_J. (represented by dashed line)
- Claim edge converts to request edge when a process requests the resource (solid line from process to resource)
- Request edge converted to an assignment edge when the resource is allocated to the process (solid line from resource to process)
- When a resource is released by a process, assignment edge reverts to a claim edge
- All resources *must be claimed a priori*.

Deadlock Avoidance *safe states (bad)*



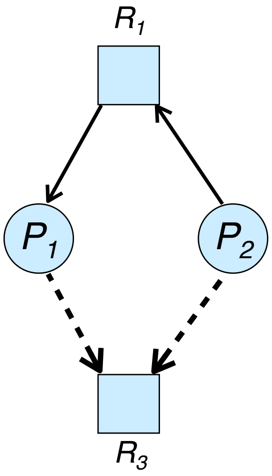
safe

unsafe
(granting R_3 to P_2 causes a cycle)

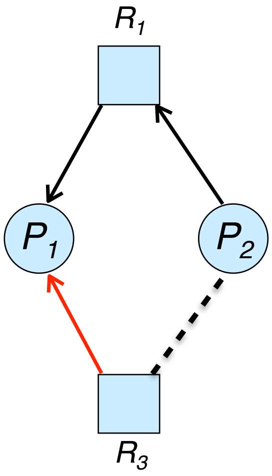
deadlock
(if P_1 converts claim to request)

Requests granted only if converting the request edge to an assignment edge does not result in a cycle

Deadlock Avoidance *safe states (good)*



safe



safe
(this would have been fine)

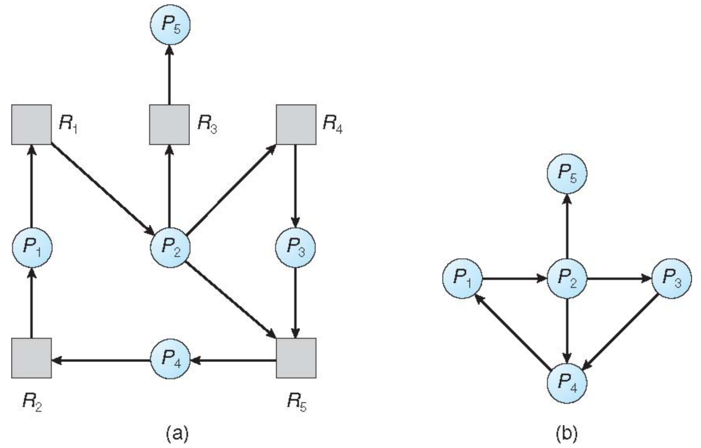
Requests granted only if converting the request edge to an assignment edge does not result in a cycle

---

Deadlock Mitigation *dealing with it*

- Maintain waits-for graph:
    - Nodes are processes
    - $P_i \to P_j$ if P_i is waiting for resource held by P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of $O(n + e)$ operations, where n,e are the number of vertices,edges in the graph

Deadlock Mitigation *dealing with it*



Resource-Allocation Graph   Corresponding wait-for graph

- Construct the waits-for graph
- Check for cycles
- Pick *any* thread of a cycle and kill it

Deadlock Mitigation *ignoring it*

"Not everything worth doing is worth doing well" - Tom West

- Consequence may be:
  - minor
  - rare

Concurrency

- Exam 1
- 26 - Concurrency
- 27 - Overview, and POSIX threads (pthreads)
- 28 - Locks
- 29 - Concurrent Data Structures
- 30 Condition Variables
- 31 - Semaphores
- 32 - Common Problems (including deadlocks)
- 33 - Event-Based Concurrency

Event-Based Concurrency *who needs threads?*

- Problems w/ thread-base concurrency:
  - software engineering:
    - missing locks, deadlocks, poor error handling
  - scheduling
    - programmer has little control over scheduling
- Event-based concurrency often used in:
  - GUI-based apps
  - internet servers (micro-services, etc.)
- Basic idea:
  - main thread waits for events:
    - do the (typically small) amount of work required
    - take actions, such as replies, scheduling other events

Event-Loop *who needs threads?*

- Basic approach:
    - wait for something (an “event”) to occur
    - perform checks based on event type
    - call event handler
- Example:

```
while(1){
      events = getEvents();
      for( e in events )
               processEvent(e); // event handler
}
```

- How to get new events?
    - `select()` or `poll()`

`select()` *as an example*

```
int select(int nfds,
         fd_set * restrict readfds,
         fd_set * restrict writefds,
         fd_set * restrict errorfds,
         struct timeval * restrict timeout);
```

- lets server know that:
    - new packet has arrived
    - room in outgoing socket
    - error conditions
    - the timeout lets `select` *poll* (“0” means synchronous)

`select()` *as an example*

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
     // open and set up a bunch of sockets (not shown)
     // main loop
     while (1) {
               // initialize the fd_set to all zero
               fd_set readFDs;
               FD_ZERO(&readFDs);

               // now set the bits for the descriptors
               // this server is interested in
               // (for simplicity, all of them from min to max)
                                   int fd;
               for (fd = minFD; fd < maxFD; fd++)
                         FD_SET(fd, &readFDs);

               // do the select
               int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);

               // check which actually have data using FD_ISSET()
               int fd;
               for (fd = minFD; fd < maxFD; fd++)
                         if (FD_ISSET(fd, &readFDs))
                                   processFD(fd);
     }
}
```

Event-Loop *simplest case*

- Why is this better?
    - assume a single CPU, no preemption
    - only needs a single thread
    - concurrency bugs can not happen
    - handling events === scheduling
- But:
    - we have many cores
    - blocking system calls!
        - if we only have a single thread, what do we do when waiting?
        - the *entire* server is waiting
- *we can not allow blocking calls*

Event-Loop *asynchronous I/O*

- Operating systems have asynchronous versions of I/O:
  - App issues I/O request and returns immediately
  - interface in Mac OS X:

```
struct aiocb {
        int aio_fildes;           /* File descriptor */
        off_t aio_offset;         /* File offset */
        volatile void *aio_buf;   /* Location of buffer */
        size_t aio_nbytes;        /* Length of transfer */
};
```

```
int aio_read(struct aiocb *aiocbp);         // start async read
```

```
int aio_error(const struct aiocb *aiocbp); // check for error
                                           // or completion
```

  - *or* use signals and signal handlers to asynchronously create appropriate *new events* when I/O completes or fails

---

Event-Loop *asynchronous I/O*

- But how to pass state to the completion handler?
  - *continuations*:
    - record state required for async I/O in some data structure
    - look it up when the I/O completes
- But the world is more complicated now:
  - single-threaded event loop:
    - i.e. `Node.js` (special cases for I/O and worker threads)
  - multi-threaded event loops:
    - Python’s `asyncio`, Java’s `ExecutorService`
    - often used w/ locks, semaphores, and msg queues
  - Actor model (Akka, Erlang):
    - each actor processes msgs asynchronously
    - actors interact via msgs instead of shared memory