Operating Systems

Deadlocks (Chapter 32)

Dr. Young-Woo Kwon

Semaphore Implementation

- Q: What if the semaphore values is negative
- A: A negative number means the number of waiters

```
void __sched down(struct semaphore *sem)
{
        unsigned long flags;

        might_sleep();
        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(sem->count > 0))
                sem->count--;
        else
                __down(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
```

```
static inline int __sched __down_common(struct semaphore *sem, long state,
                                        long timeout)
{
        int ret;

        trace_contention_begin(lock: sem, flags: 0);
        ret = ___down_common(sem, state, timeout);
        trace_contention_end(lock: sem, ret);

        return ret;
}
```

```
static inline int __sched ___down_common(struct semaphore *sem, long state,
                                         long timeout)
{
        struct semaphore_waiter waiter;

        list_add_tail(new: &waiter.list, head: &sem->wait_list);
        waiter.task = current;
        waiter.up = false;

        for (;;) {
                if (signal_pending_state(state, current))
                        goto interrupted;
                if (unlikely(timeout <= 0))
                        goto timed_out;
                __set_current_state(state);
                raw_spin_unlock_irq(&sem->lock);
                timeout = schedule_timeout(timeout);
                raw_spin_lock_irq(&sem->lock);
                if (waiter.up)
                        return 0;
        }

 timed_out:
        list_del(entry: &waiter.list);
        return -ETIME;

 interrupted:
        list_del(entry: &waiter.list);
        return -EINTR;
}
```

Semaphore Implementation

- Semaphore up()

```c
static noinline void __sched __up(struct semaphore *sem)
{
	struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list,
						struct semaphore_waiter, list);
	list_del(entry: &waiter->list);
	waiter->up = true;
	wake_up_process(tsk: waiter->task);
}
```

32. Common Concurrency Problems.

Bugs in Modern Applications

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Non-Deadlock Bugs

- Make up a majority of concurrency bugs.
- Two major types of non deadlock bugs:
  - Atomicity violation
  - Order violation

Atomicity-Violation Bugs

- The desired **serializability** among multiple memory accesses is *violated*.
  - Simple Example found in MySQL:
    - Two different threads access the field `proc_info` in the `struct thd`.

```
Thread1::
if(thd->proc_info){
   …
   fputs(thd->proc_info , …);
   …
}

Thread2::
thd->proc_info = NULL;
```

Atomicity-Violation Bugs (Cont.)

- **Solution**: Simply add locks around the shared-variable references.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Thread1::
pthread_mutex_lock(&lock);
if(thd->proc_info){
    …
    fputs(thd->proc_info , …);
    …
}
pthread_mutex_unlock(&lock);

Thread2::
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```

Order-Violation Bugs

- The **desired order** between two memory accesses is <u>flipped</u>.
  - i.e., **`A`** should always be executed before **`B`**, but the order is not enforced during execution.
  - **Example**:
    - The code in Thread2 seems to assume that the variable `mThread` has already been *initialized* (and is not `NULL`).

```
Thread1::
void init(){
    mThread = PR_CreateThread(mMain, …);
}

Thread2::
void mMain(…){
    mState = mThread->State
}
```

Order-Violation Bugs (Cont.)

- **Solution**: Enforce ordering using condition variables

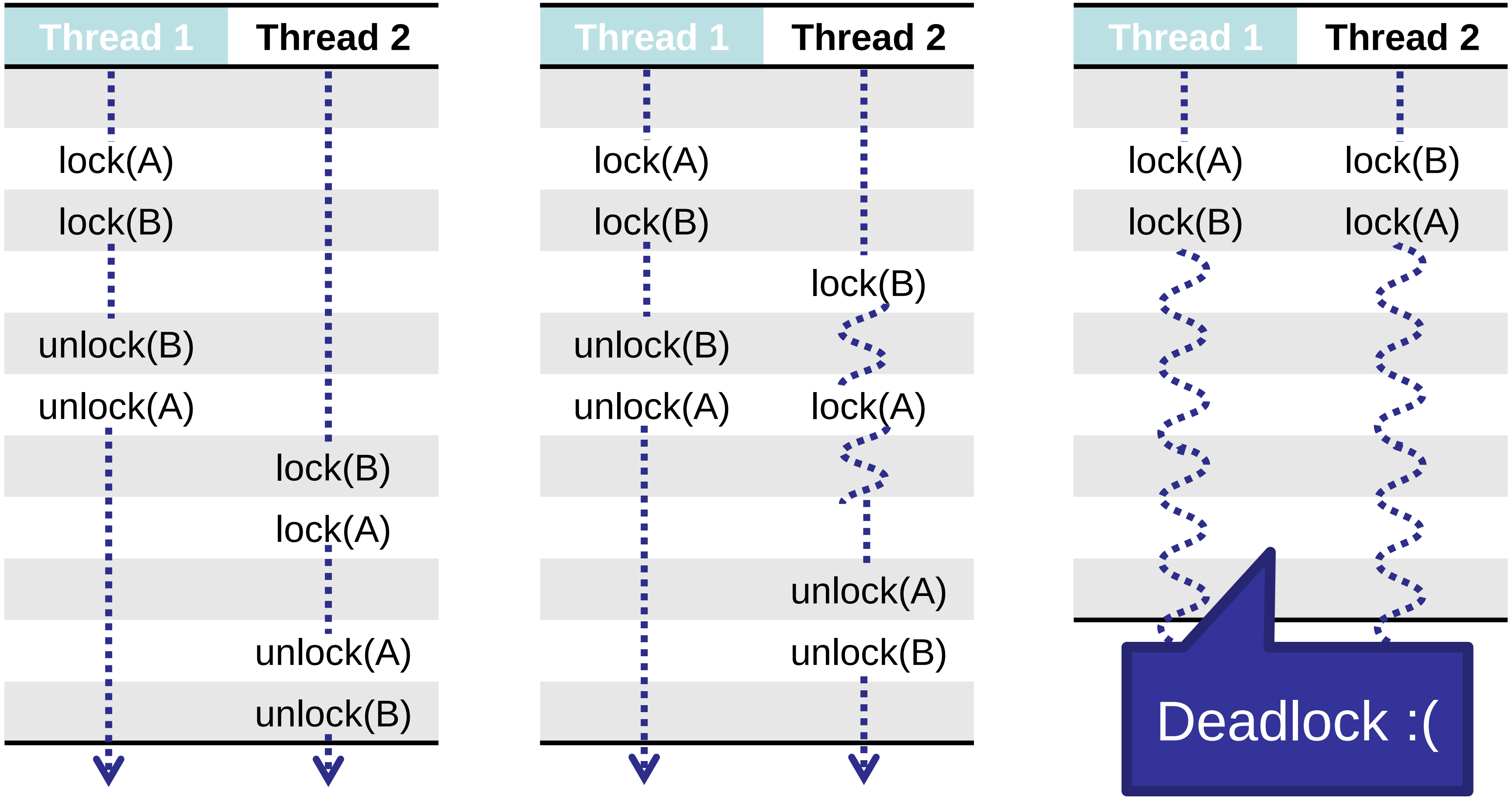
```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
int mtInit = 0;

Thread 1::
void init(){
    …
    mThread = PR_CreateThread(mMain,…);

    // signal that the thread has been created.
    pthread_mutex_lock(&mtLock);
    mtInit = 1;
    pthread_cond_signal(&mtCond);
    pthread_mutex_unlock(&mtLock);
    …
}
```

Order-Violation Bugs (Cont.)

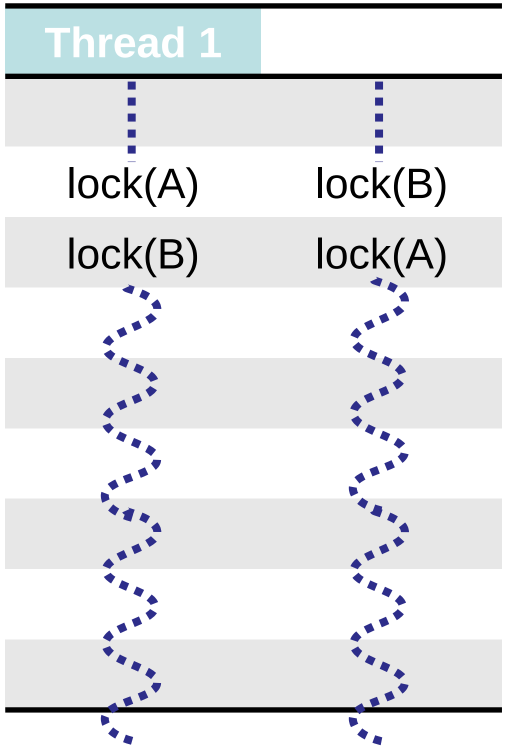
```
Thread2::
void mMain(…){
    // wait for the thread to be initialized …
    pthread_mutex_lock(&mtLock);
    while(mtInit == 0)
            pthread_cond_wait(&mtCond, &mtLock);
    pthread_mutex_unlock(&mtLock);

    mState = mThread->State;
    …
}
```

Deadlock Bugs

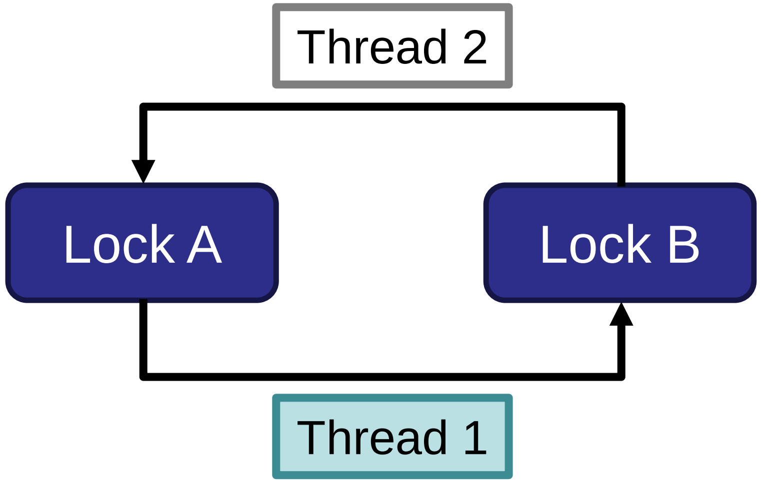
	Thread 1	Thread 2
mutex A	lock A	lock B
mutex B	lock B	lock A
	// do something	// do something
	unlock B	unlock A
	unlock A	unlock B



Deadlock



- Simple example of circular waiting
  - Thread 1 holds lock *a*, waits on lock *b*
  - Thread 2 holds lock *b*, waits on lock *a*



Why Do Deadlocks Occur?

- Reason 1:
  - In large code bases, **complex dependencies** arise between components.

- Reason 2:
  - Due to the nature of **encapsulation**
    - Hide details of implementations and make software easier to build in a modular way.
    - Such modularity *does not mesh* well with locking.

Conditions for Deadlock

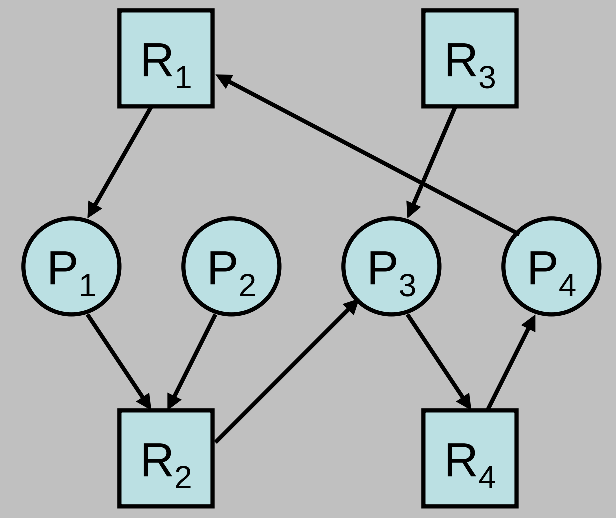
- Four conditions need to hold for a deadlock to occur.

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

- If any of these four conditions are not met, **deadlock cannot occur**.
- But, one more issue:
  - Buggy programming: programmer forgets to release one or more resources

Deadlocks, more formally

- 4 necessary conditions
  1) Exclusive Access
  2) Hold and Wait
  3) No Preemption
  4) Circular Wait
  - Note that cond 1-3 represent things that are normally desirable or required
- Will look at strategies to
  - Detect & break deadlocks
  - Prevent
  - Avoid



Resource Allocation Graph

$R \rightarrow P$ Assignment

$P \rightarrow R$ Request

Detect and Recover

- Allow deadlock to occasionally occur and then take some action.
  - **Example**: if an OS froze, you would reboot it.

- Many database systems employ *deadlock detection* and *recovery technique*.
  - A deadlock detector **runs periodically**.
  - Building a **resource graph** and checking it for **cycles**.
  - In deadlock, the system **need to be restarted**.

Deadlock Detection

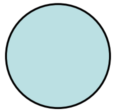
- Idea: Look for circularity in resource allocation graph
  - Q.: How do you find out if a directed graph has a cycle?
- Can be done eagerly
  - on every resource acquisition/release, resource allocation graph is updated & tested
- or lazily
  - when all threads are blocked & deadlock is suspected, build graph & test

Resource-Allocation Graph

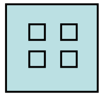
- A set of vertices *V* and a set of edges *E*.
  - V is partitioned into two types:
    - P = {P1, P2, …, Pn}, the set consisting of all the processes in the system
    - R = {R1, R2, …, Rm}, the set consisting of all resource types in the system
  - request edge – directed edge Pi $\rightarrow$ Rj
  - assignment edge – directed edge Rj $\rightarrow$ Pi

Resource-Allocation Graph (Cont.)

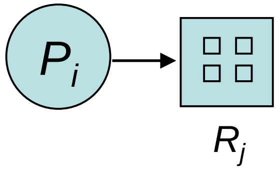
- Process



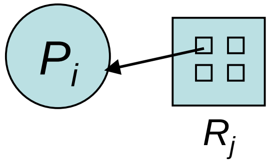
- Resource Type with 4 instances



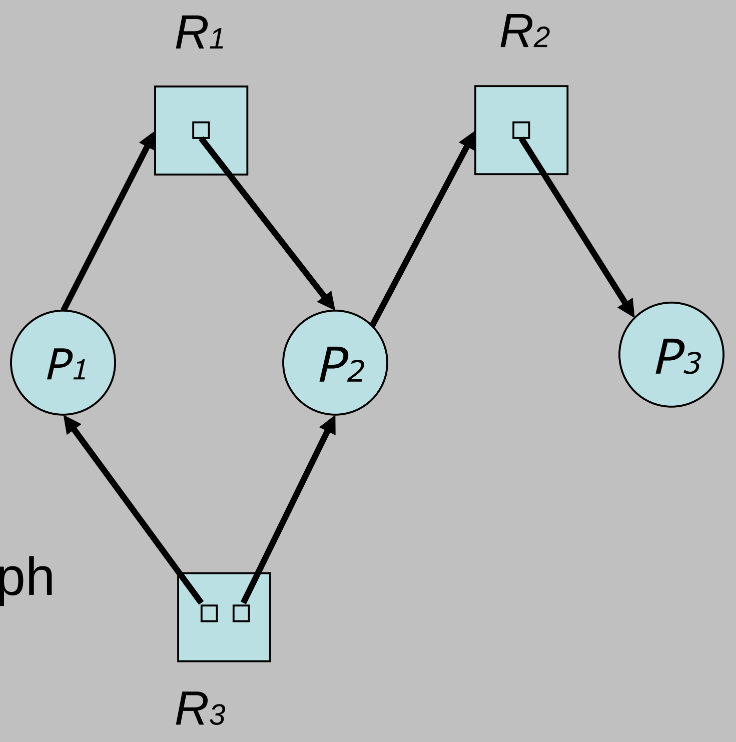
- P_i requests instance of R_j



- P_i is holding an instance of R_j

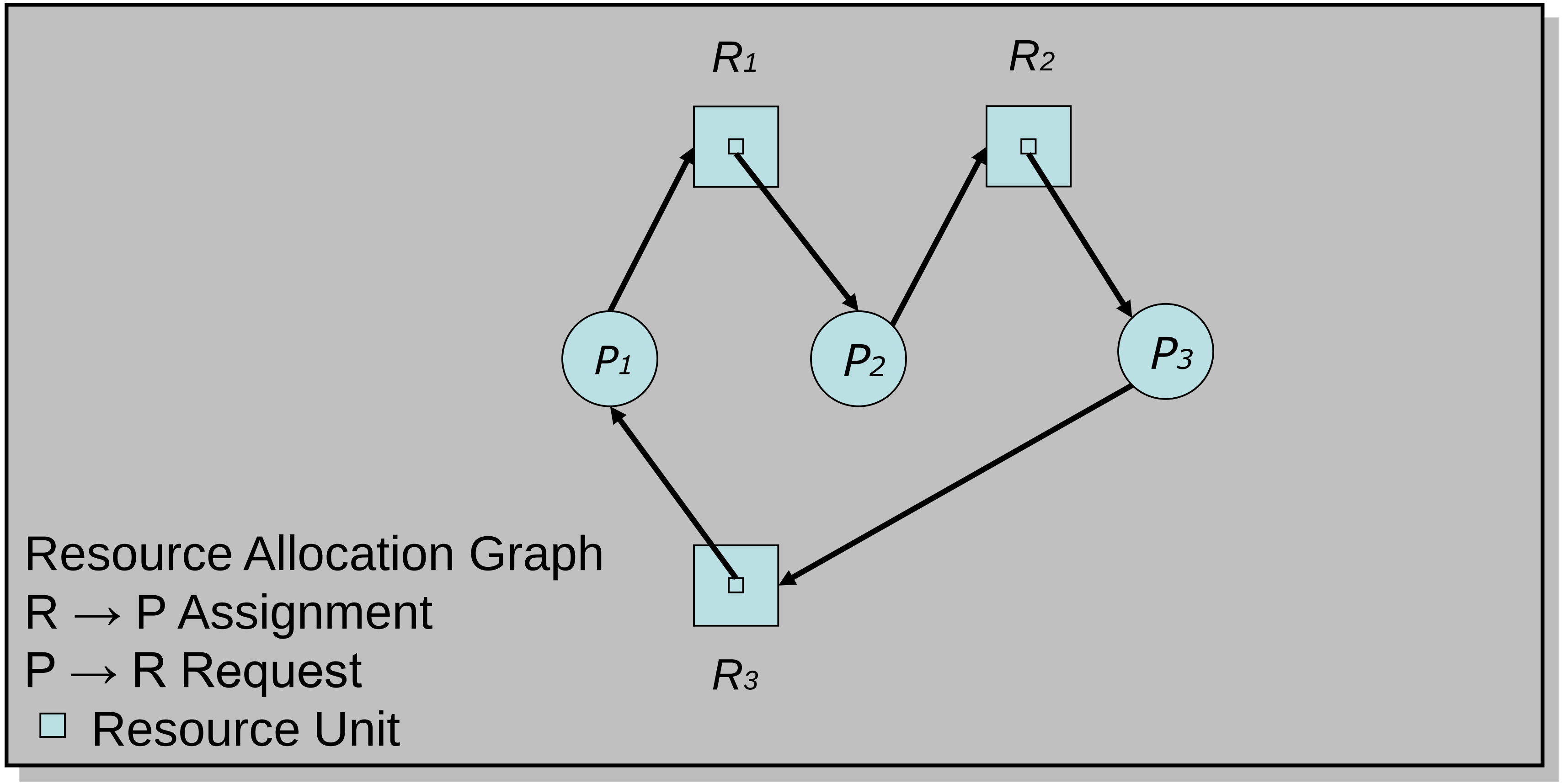


Example of a Resource Allocation Graph

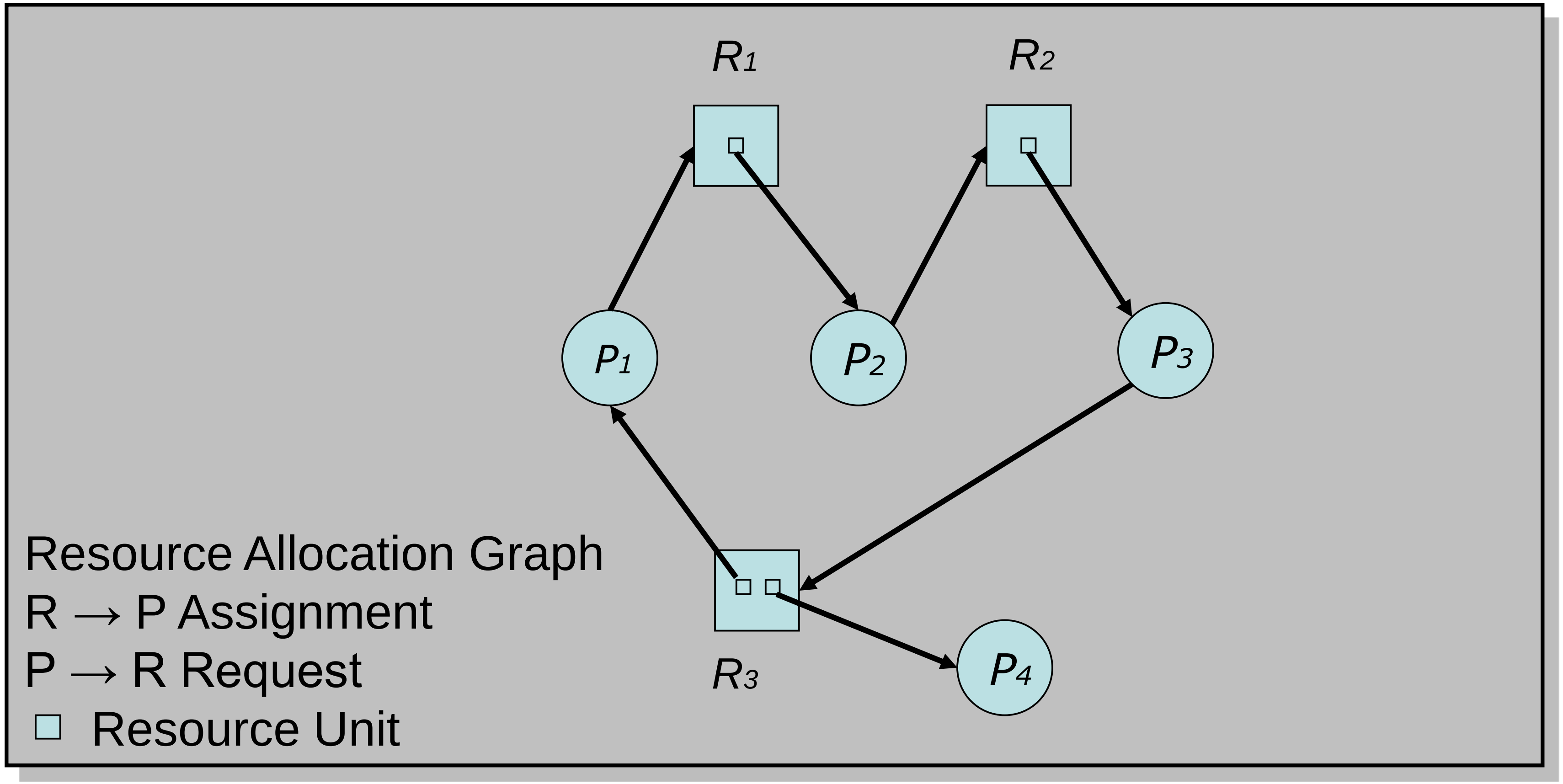


Resource Allocation Graph
R → P Assignment
P → R Request
□ Resource Unit

Resource Allocation Graph with a Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



Deadlock Detection

- Basic facts
  - If each resource has exactly one unit, deadlock iff cycle
  - If each resource has multiple units, existence of cycle may or may not mean deadlock

- Q.: What to do once deadlock is detected?

Deadlock Recovery

Increasing Severity

- Preempt resources (if possible)
- Back processes up to a checkpoint
  - Requires checkpointing or transactions (typically expensive)
- Kill processes involved until deadlock is resolved
- Kill all processes involved
- Reboot

Prevention – Circular Wait

- Provide a total ordering on lock acquisition
  - This approach requires *careful design* of global locking strategies.
- Provide a partial ordering in complex systems

- **Example**:
  - There are two locks in the system (L1 and L2)
  - We can prevent deadlock by always acquiring L1 before L2.

Prevention – Hold-and-wait

- Acquire all locks at once, atomically.

```
lock(prevention);
lock(L1);
lock(L2);
…
unlock(prevention);
```

  – This code guarantees that **no untimely thread switch can occur** *in the midst of* lock acquisition.

  – **Problem**:

    - Require us to know when calling a routine exactly which locks must be held and to acquire them ahead of time.
    - Decrease *concurrency*

Prevention – No Preemption

- Take resource away from process
  - Difficult: how should process react?
- Virtualize resource so it can be taken away
  - Requires saving & restoring resource’s state

Prevention – No Preemption

- **Multiple lock acquisition** often gets us into trouble because when waiting for one lock we are holding another.
- `trylock()`
  - Used to build a *deadlock-free*, *ordering-robust* lock acquisition protocol.
  - Grab the lock (if it is available).
  - Or, return -1: you should try again later.

```
top:
    lock(L1);
    if( tryLock(L2) == -1 ){
            unlock(L1);
            goto top;
    }
```

Prevention – No Preemption (Cont.)

- livelock
  - Both systems are running through the code sequence *over and over again*.
  - Progress is not being made.
  - Solution:
    - Add **a random delay** before looping back and trying the entire thing over again.

Prevention – Mutual Exclusion

- wait-free
  - Using powerful hardware instruction.
  - You can build data structures in a manner that *does not require* explicit locking.

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

Prevention – Mutual Exclusion (Cont.)

- We now wanted to atomically increment a value by a certain amount:

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

  - Repeatedly tries to update the value to *the new amount* and uses the `compare-and-swap` to do so.
  - **No lock** is acquired
  - **No deadlock** can arise
  - **livelock** is still a possibility.

Prevention – Mutual Exclusion (Cont.)

- **Solution**:
  - Surrounding this code with a lock acquire and release.

```
void insert(int value){
     node_t * n = malloc(sizeof(node_t));
     assert( n != NULL );
     n->value = value ;
     lock(listlock); // begin critical section
     n->next  = head;
     head     = n;
     unlock(listlock) ;  //end critical section
}
```

  - wait-free manner using the `compare-and-swap` instruction

```
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));
}
```

Deadlock Avoidance via Scheduling

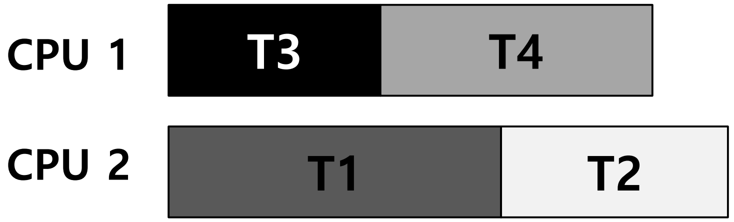
- Deadlock Avoidance
  - Get the information about the locks various threads might grab during their execution.
  - schedule the threads in a way <u>to guarantee</u> no deadlock can occur.
- In some scenarios, deadlock avoidance is preferable.
- Problem: Global knowledge is required.

Example of Deadlock Avoidance via Scheduling (1)

- We have two processors and four threads.
  - Lock acquisition demands of the threads:

	T1	T2	T3	T4
L1	**yes**	**yes**	no	no
L2	**yes**	**yes**	yes	no

  - A smart scheduler could compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise.

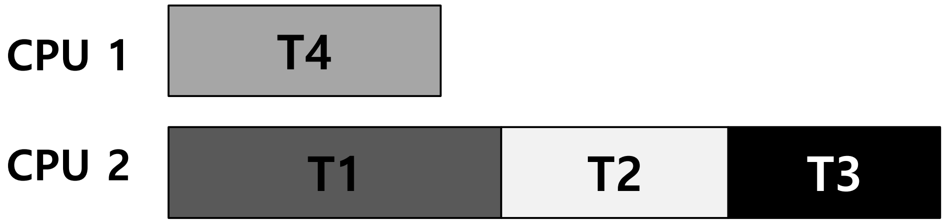


Example of Deadlock Avoidance via Scheduling (2)

- More contention for the same resources

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

  – A possible schedule that guarantees that *no deadlock* could ever occur.



    - The total time to complete the jobs is lengthened considerably.

Deadlock Avoidance via Scheduling

- Banker's algorithm
  - How much of each resource each process could possibly request
  - How much of each resource each process is currently holding
  - How much of each resource the system has available

Total resources in system:

A	B	C	D
6	5	7	6

Processes (maximum resources required by P):

	A	B	C	D
P1	3	3	2	2
P2	1	2	3	4
P3	1	3	5	0

Processes (currently allocated resources):

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	1	0
Alloc.	3	4	6	4

Need = Max - current:

	A	B	C	D
P1	2	1	0	1
P2	0	2	0	1
P3	0	1	4	0

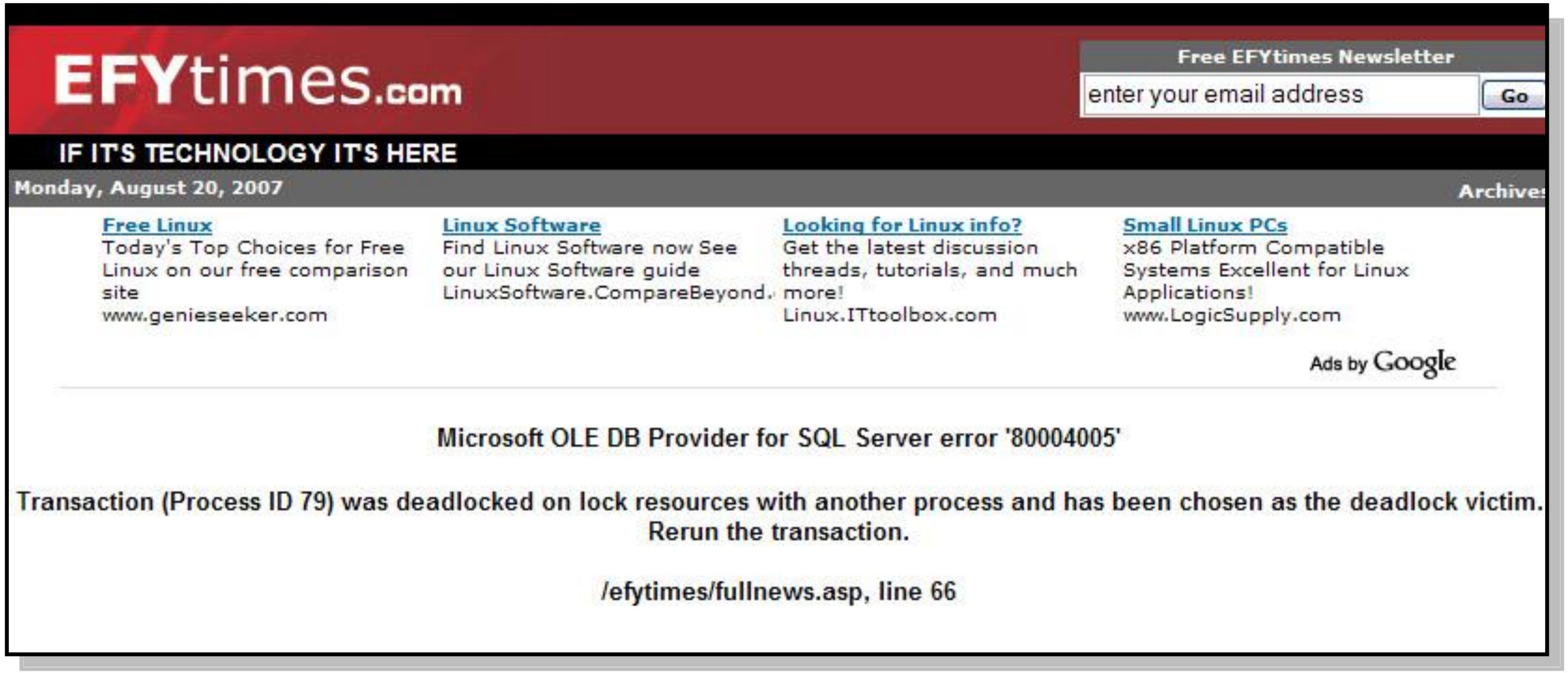
Available system resources (Total – Allocated)

A	B	C	D
3	1	1	2

Request will only be granted under below condition.

1. If request made by process is less than equal to max need to that process.
2. If request made by process is less than equal to freely available resource in the system.

Deadlock In The Real World



Deadlock in the Real World

- Most common strategy of handling deadlock
  - Test phase: fix all deadlocks detected during testing
  - Deploy phase: if deadlock happens, kill and rerun (easy!)
    - If it happens too often, or reproducibly, add deadlock detection code
- Weigh cost of **preventing** vs cost of (re-) **occurring**
- Static analysis tools detects some kinds of deadlocks before they occur
  - Idea: monitor order in which locks are taken, flag if not consistent lock order

Deadlock vs. Starvation

- Deadlock:
  - No matter which policy the scheduler chooses, there is **no possible way for processes** to make forward progress
- Starvation:
  - There is **a possible way in which threads can make possible forward** progress, but the scheduler doesn't choose it
    - Example: strict priority scheduler will never scheduler lower priority threads as long as higher-priority thread is READY
    - Example: naïve reader/writer lock: starvation may occur by "bad luck"

How many locks should I use?

- Could use one lock for all shared variables
  - Disadvantage: if a thread holding the lock blocks, no other thread can access *any* shared variable, even unrelated ones

- Ideally, want fine-grained locking
  - One lock only protects one (or a small set of) variables – how to pick that set?

Multiple locks, the wrong way

```
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */

static struct lock alloclock; /* Protects allocations */
static struct lock freelock; /* Protects deallocations */
```

```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&alloclock);
    b = alloc_block_from_freelist();
    insert_into_usedlist(&usedlist, b);
    lock_release(&alloclock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&freelock);
    list_remove(&b->elem);
    coalesce_into_freelist(&freelist, b);
    lock_release(&freelock);
}
```

Wrong: Allocating thread & deallocating thread could collide

Multiple locks, 2nd try

```
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */

static struct lock usedlock; /* Protects usedlist */
static struct lock freelock;   /* Protects freelist */
```

```
void *mem_alloc(...)
{
    block *b;
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    lock_acquire(&usedlock);
    insert_into_usedlist(&usedlist, b);
    lock_release(&freelock);
    lock_release(&usedlock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&usedlock);
    list_remove(&b->elem);
    lock_acquire(&freelock);
    coalesce_into_freelist(&freelist, b);
    lock_release(&usedlock);
    lock_release(&freelock);
```

Also wrong: deadlock!
Always acquire multiple locks in same order -
Or don't hold them simultaneously

Multiple locks, correct (1)

```
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */

static struct lock usedlock; /* Protects usedlist */
static struct lock freelock;   /* Protects freelist */
```

```
void *mem_alloc(…)
{
    block *b;
    lock_acquire(&usedlock);
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    insert_into_usedlist(&usedlist, b);
    lock_release(&freelock);
    lock_release(&usedlock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&usedlock);
    lock_acquire(&freelock);
    list_remove(&b->elem);
    coalesce_into_freelist(&freelist, b);
    lock_release(&freelock);
    lock_release(&usedlock);
```

Correct, but inefficient!
Locks are always held simultaneously,
one lock would suffice

Multiple locks, correct (2)

```
static struct list usedlist; /* List of used blocks */
static struct list freelist;  /* List of free blocks */

static struct lock usedlock; /* Protects usedlist */
static struct lock freelock;   /* Protects freelist */
```

```
void *mem_alloc(…)
{
    block *b;
    lock_acquire(&freelock);
    b = alloc_block_from_freelist();
    lock_release(&freelock);
    lock_acquire(&usedlock);
    insert_into_usedlist(&usedlist, b);
    lock_release(&usedlock);
    return b->data;
}
```

```
void mem_free(block *b)
{
    lock_acquire(&usedlock);
    list_remove(&b->elem);
    lock_release(&usedlock);
    lock_acquire(&freelock);
    coalesce_into_freelist(&freelist, b);
    lock_release(&freelock);
}
```

Conclusion

- Choosing which lock should protect which shared variable(s) is not easy – must weigh:
  - Whether all variables are always accessed together (use one lock if so)
  - Whether code inside critical section may block (if not, no throughput gain from fine-grained locking on uniprocessor)
  - Whether there is a consistency requirement if multiple variables are accessed in related sequence (must hold single lock if so)
  - Cost of multiple calls to lock/unlock (increasing parallelism advantages may be offset by those costs)