Concurrency

- On multiprocessors, several threads can execute simultaneously, one on each processor.
- On uniprocessors, only one thread executes at a time. However, because of preemption and timesharing, threads appear to run concurrently.

Concurrency and synchronization are important even on uniprocessors.

Thread Synchronization

- Concurrent threads can interact with each other in a variety of ways:
  - Threads share access, though the operating system, to system devices (more on this later . . .)
  - Threads may share access to program data, e.g., global variables.
- A common synchronization problem is to enforce *mutual exclusion*, which means making sure that only one thread at a time uses a shared object, e.g., a variable or a device.
- The part of a program in which the shared object is accessed is called a *critical section*.

Critical Section Example (Part 1)

```
int list_remove_front(list *lp) {
   int num;
   list_element *element;
   assert(!is_empty(lp));
   element = lp->first;
   num = lp->first->item;
   if (lp->first == lp->last) {
     lp->first = lp->last = NULL;
   } else {
     lp->first = element->next;
   }
   lp->num_in_list--;
   free element;
   return num;
}
```

The `list_remove_front` function is a critical section. It may not work properly if two threads call it at the same time on the same `list`. (Why?)

Critical Section Example (Part 2)

```
void list_append(list *lp, int new_item) {
   list_element *element = malloc (list_element));
   element->item = new_item
   assert(!is_in_list(lp, new_item));
   if (is_empty(lp)) {
     lp->first = element; lp->last = element;
   } else {
     lp->last->next = element; lp->last = element;
   }
   lp->num_in_list++;
}
```

The `list_append` function is part of the same critical section as `list_remove_front`. It may not work properly if two threads call it at the same time, or if a thread calls it while another has called `list_remove_front`

Enforcing Mutual Exclusion

- mutual exclusion algorithms ensure that only one thread at a time executes the code in a critical section
- several techniques for enforcing mutual exclusion
  - exploit special hardware-specific machine instructions, e.g., *test-and-set* or *compare-and-swap*, that are intended for this purpose
  - use mutual exclusion algorithms, e.g., *Peterson's algorithm*, that rely only on atomic loads and stores
  - control interrupts to ensure that threads are not preempted while they are executing a critical section

Disabling Interrupts

- On a uniprocessor, only one thread at a time is actually running.
- If the running thread is executing a critical section, mutual exclusion may be violated if
  1. the running thread is preempted (or voluntarily yields) while it is in the critical section, and
  2. the scheduler chooses a different thread to run, and this new thread enters the same critical section that the preempted thread was in
- Since preemption is caused by timer interrupts, mutual exclusion can be enforced by disabling timer interrupts before a thread enters the critical section, and re-enabling them when the thread leaves the critical section.

This is the way that the OS/161 kernel enforces mutual exclusion. There is a simple interface (`splhigh()`, `spl0()`, `splx()`) for disabling and enabling interrupts. See `kern/arch/mips/include/spl.h`.

Pros and Cons of Disabling Interrupts

- advantages:
  - does not require any hardware-specific synchronization instructions
  - works for any number of concurrent threads
- disadvantages:
  - indiscriminate: prevents all preemption, not just preemption that would threaten the critical section
  - ignoring timer interrupts has side effects, e.g., kernel unaware of passage of time. (Worse, OS/161's `splhigh()` disables *all* interrupts, not just timer interrupts.) Keep critical sections *short* to minimize these problems.
  - will not enforce mutual exclusion on multiprocessors (why??)

Peterson's Mutual Exclusion Algorithm

```
/* shared variables */
/* note: one flag array and turn variable */
/* for each critical section */
boolean flag[2];  /* shared, initially false */
int turn;         /* shared */

flag[i] = true;   /* in one process, $i = 0$ and $j = 1$ */
turn = j;         /* in the other, $i = 1$ and $j = 0$ */
while (flag[j] && turn == j) { }  /* busy wait */

 critical section    /* e.g., call to list_remove_front */

flag[i] = false;
```

Ensures mutual exclusion and avoids starvation, but works only for two processes. (Why?)

Mutual Exclusion Using Special Instructions

- Software solutions to the critical section problem (e.g., Peterson's algorithm) assume only atomic load and atomic store.
- Simpler algorithms are possible if more complex *atomic* operations are supported by the hardware. For example:

  Test and Set: set the value of a variable, and return the old value

  Swap: swap the values of two variables
- On uniprocessors, mutual exclusion can also be achieved by disabling interrupts during the critical section. (Normally, user programs cannot do this, but the kernel can.)

Hardware-Specific Synchronization Instructions

- a test-and-set instruction *atomically* sets the value of a specified memory location and either
  - places that memory location's *old* value into a register, or
  - checks a condition against the memory location's old value and records the result of the check in a register
- for presentation purposes, we will abstract such an instruction as a function `TestAndSet(address,value)`, which takes a memory location (`address`) and a value as parameters. It atomically stores `value` at the memory location specified by `address` and returns the previous value stored at that address

A Spin Lock Using Test-And-Set

- a test-and-set instruction can be used to enforce mutual exclusion
- for each critical section, define a `lock` variable

  ```
  boolean lock;  /* shared, initially false */
  ```

  We will use the lock variable to keep track of whether there is a thread in the critical section, in which case the value of `lock` will be `true`
- before a thread can enter the critical section, it does the following:

  ```
  while (TestAndSet(&lock,true)) { }  /* busy-wait */
  ```
- when the thread leaves the critical section, it does

  ```
  lock = false;
  ```
- this enforces mutual exclusion (why?), but starvation is a possibility

---

This construct is sometimes known as a *spin lock*, since a thread "spins" in the while loop until the critical section is free. Spin locks are widely used on multiprocessors.

Semaphores

- A semaphore is a synchronization primitive that can be used to enforce mutual exclusion requirements. It can also be used to solve other kinds of synchronization problems.
- A semaphore is an object that has an integer value, and that supports two operations:

  P: if the semaphore value is greater than 0, decrement the value. Otherwise, wait until the value is greater than 0 and then decrement it.

  V: increment the value of the semaphore
- Two kinds of semaphores:

  counting semaphores: can take on any non-negative value

  binary semaphores: take on only the values 0 and 1. (V on a binary semaphore with value 1 has no effect.)

By definition, the P and V operations of a semaphore are *atomic*.

OS/161 Semaphores

```
struct semaphore {
  char *name;
  volatile int count;
};

struct semaphore *sem_create(const char *name,
     int initial_count);
void P(struct semaphore *);
void V(struct semaphore *);
void sem_destroy(struct semaphore *);
```

see

- `kern/include/synch.h`
- `kern/thread/synch.c`

OS/161 Semaphores: P()

```
void
P(struct semaphore *sem)
{
  int spl;
  assert(sem != NULL);

  /*
   * May not block in an interrupt handler.
   * For robustness, always check, even if we can actually
   * complete the P without blocking.
   */
  assert(in_interrupt==0);

  spl = splhigh();
  while (sem->count==0) {
    thread_sleep(sem);
  }
  assert(sem->count>0);
  sem->count--;
  splx(spl);
}
```

Thread Blocking

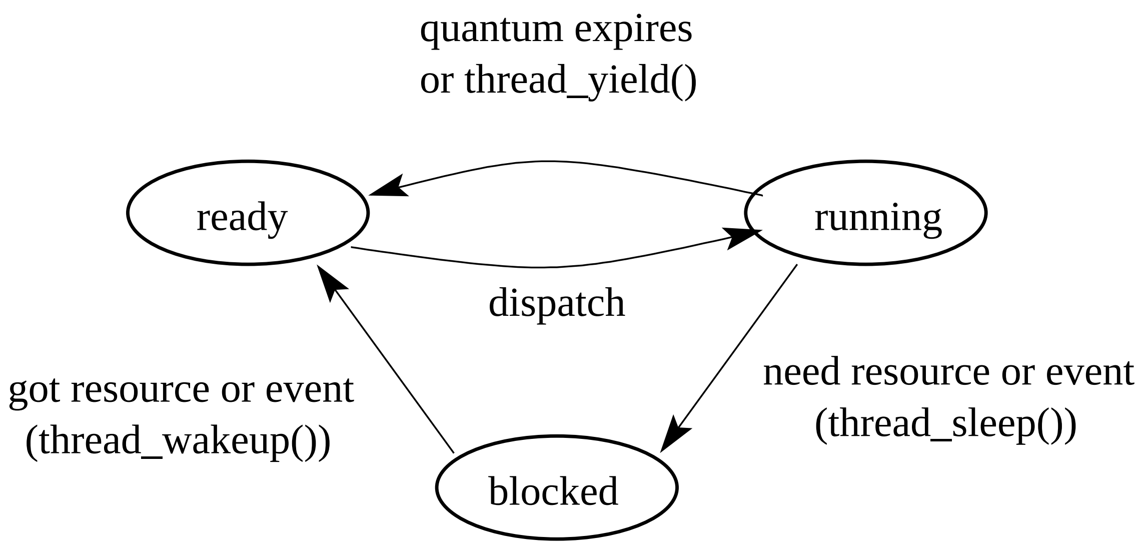
- Sometimes a thread will need to wait for an event. One example is on the previous slide: a thread that attempts a P() operation on a zero-valued semaphore must wait until the semaphore's value becomes positive.
- other examples that we will see later on:
  - wait for data from a (relatively) slow device
  - wait for input from a keyboard
  - wait for busy device to become idle
- In these circumstances, we do not want the thread to run, since it cannot do anything useful.
- To handle this, the thread scheduler can *block* threads.

Thread Blocking in OS/161

- OS/161 thread library functions:
  - `void thread_sleep(const void *addr)`
    - blocks the calling thread on address `addr`
  - `void thread_wakeup(const void *addr)`
    - unblock threads that are sleeping on address `addr`
- `thread_sleep()` is much like `thread_yield()`. The calling thread voluntarily gives up the CPU, the scheduler chooses a new thread to run, and dispatches the new thread. However
  - after a `thread_yield()`, the calling thread is *ready* to run again as soon as it is chosen by the scheduler
  - after a `thread_sleep()`, the calling thread is blocked, and should not be scheduled to run again until after it has been explicitly unblocked by a call to `thread_wakeup()`.

Thread States

- a very simple thread state transition diagram



- the states:

  running: currently executing

  ready: ready to execute

  blocked: waiting for something, so not ready to execute.

OS/161 Semaphores: V() kern/thread/synch.c

```
void
V(struct semaphore *sem)
{
  int spl;
  assert(sem != NULL);
  spl = splhigh();
  sem->count++;
  assert(sem->count>0);
  thread_wakeup(sem);
  splx(spl);
}
```

Mutual Exclusion Using a Semaphore

```
struct semaphore *s;
s = sem_create("MySem1", 1); /* initial value is 1 */

P(s);  /* do this before entering critical section */

   critical section /* e.g., call to list_remove_front */

V(s);  /* do this after leaving critical section */
```

Producer/Consumer Synchronization

- suppose we have threads that add items to a list (producers) and threads the remove items from the list (consumers)
- suppose we want to ensure that consumers do not consume if the list is empty - instead they must wait until the list has something in it
- this requires synchronization between consumers and producers
- semaphores can provide the necessary synchronization, as shown on the next slide

Producer/Consumer Synchronization using Semaphores

```
struct semaphore *s;
s = sem_create("Items", 0); /* initial value is 0 */

Producer's Pseudo-code:
  add item to the list (call list_append())
  V(s);

Consumer's Pseudo-code:
  P(s);
  remove item from the list (call list_remove_front())
```

The Items semaphore does not enforce mutual exclusion on the list. If we want mutual exclusion, we can also use semaphores to enforce it. (How?)

Bounded Buffer Producer/Consumer Synchronization

- suppose we add one more requirement: the number of items in the list should not exceed `N`
- producers that try to add items when the list is full should be made to wait until the list is no longer full
- We can use an additional semaphore to enforce this new constraint:
  - semaphore `Full` is used to enforce the constraint that producers should not produce if the list is full
  - semaphore `Empty` is used to enforce the constraint that consumers should not consume if the list is empty

```
struct semaphore *full;
struct semaphore *empty;
full = sem_create("Full", 0);   /* initial value = 0 */
empty = sem_create("Empty", N); /* initial value = N */
```

Bounded Buffer Producer/Consumer Synchronization with Semaphores

```
Producer's Pseudo-code:
  P(empty);
  add item to the list (call list_append())
  V(full);

Consumer's Pseudo-code:
  P(full);
  remove item from the list (call list_remove_front())
  V(empty);
```

OS/161 Locks

- OS/161 also uses a synchronization primitive called a *lock*. Locks are intended to be used to enforce mutual exclusion.

```
struct lock *mylock = lock_create("LockName");

lock_aquire(mylock);
    critical section /* e.g., call to list_remove_front */
lock_release(mylock);
```

- A lock is similar to a binary semaphore with an initial value of 1. However, locks also enforce an additional constraint: the thread that releases a lock must be the same thread that most recently acquired it.
- The system enforces this additional constraint to help ensure that locks are used as intended.

Condition Variables

- OS/161 supports another common synchronization primitive: *condition variables*
- each condition variable is intended to work together with a lock: condition variables are only used *from within the critical section that is protected by the lock*
- three operations are possible on a condition variable:

  wait: this causes the calling thread to block, and it releases the lock associated with the condition variable

  signal: if threads are blocked on the signaled condition variable, then one of those threads is unblocked

  broadcast: like signal, but unblocks all threads that are blocked on the condition variable

Waiting on Condition Variables

- when a blocked thread is unblocked (by `signal` or `broadcast`), it reacquires the lock before returning from the `wait` call
- a thread is in the critical section when it calls `wait`, and it will be in the critical section when `wait` returns. However, in between the call and the return, while the caller is blocked, the caller is out of the critical section, and other threads may enter.
- In particular, the thread that calls `signal` (or `broadcast`) to wake up the waiting thread will itself be in the critical section when it signals. The waiting thread will have to wait (at least) until the signaller releases the lock before it can unblock and return from the `wait` call.

This describes Mesa-style condition variables, which are used in OS/161. There are alternative condition variable semantics (Hoare semantics), which differ from the semantics described here.

Using Condition Variables

- Condition variables get their name because they allow threads to wait for arbitrary conditions to become true inside of a critical section.
- Normally, each condition variable corresponds to a particular condition that is of interest to an application. For example, in the bounded buffer producer/consumer example on the following slides, the two conditions are:
  - $count > 0$ (condition variable `notempty`)
  - $count < N$ (condition variable `notfull`)
- when a condition is not true, a thread can `wait` on the corresponding condition variable until it becomes true
- when a thread detects that a condition it true, it uses `signal` or `broadcast` to notify any threads that may be waiting

Note that signalling (or broadcasting to) a condition variable that has no waiters has *no effect*. Signals do not accumulate.

Bounded Buffer Producer Using Condition Variables

```
int count = 0;  /* must initially be 0 */
struct lock *mutex;     /* for mutual exclusion */
struct cv *notfull, *notempty; /* condition variables */

/* Initialization Note: the lock and cv's must be created
 * using lock_create() and cv_create() before Produce()
 * and Consume() are called */

Produce(item) {
  lock_acquire(mutex);
  while (count == N) {
     cv_wait(notfull, mutex);
  }
  add item to buffer (call list_append())
  count = count + 1;
  cv_signal(notempty, mutex);
  lock_release(mutex);
}
```

Bounded Buffer Consumer Using Condition Variables

```
Consume() {
  lock_acquire(mutex);
  while (count == 0) {
     cv_wait(notempty, mutex);
  }
  remove item from buffer (call list_remove_front())
  count = count - 1;
  cv_signal(notfull, mutex);
  lock_release(mutex);
}
```

Both Produce() and Consume() call cv_wait() inside of a `while` loop. Why?

Monitors

- Condition variables are derived from *monitors*. A monitor is a programming language construct that provides synchronized access to shared data. Monitors have appeared in many languages, e.g., Ada, Mesa, Java
- a monitor is essentially an object with special concurrency semantics
- it is an object, meaning
  - it has data elements
  - the data elements are encapsulated by a set of methods, which are the only functions that directly access the object's data elements
- only *one* monitor method may be active at a time, i.e., the monitor methods (together) form a critical section
  - if two threads attempt to execute methods at the same time, one will be blocked until the other finishes
- inside a monitor, so called *condition variables* can be declared and used

Monitors in OS/161

- The C language, in which OS/161 is written, does not support monitors.
- However, programming convention and OS/161 locks and condition variables can be used to provide monitor-like behavior for shared kernel data structures:
  - define a C structure to implement the object's data elements
  - define a set of C functions to manipulate that structure (these are the object "methods")
  - ensure that only those functions directly manipulate the structure,
  - create an OS/161 lock to enforce mutual exclusion
  - ensure that each access method acquires the lock when it starts and releases the lock when it finishes
  - if desired, define one or more condition variables and use them within the methods.

Deadlocks

- Suppose there are two threads and two locks, `lockA` and `lockB`, both intiatially unlocked.
- Suppose the following sequence of events occurs
  1. Thread 1 does `lock_acquire(lockA)`.
  2. Thread 2 does `lock_acquire(lockB)`.
  3. Thread 1 does `lock_acquire(lockB)` and blocks, because `lockB` is held by thread 2.
  4. Thread 2 does `lock_acquire(lockA)` and blocks, because `lockA` is held by thread 1.

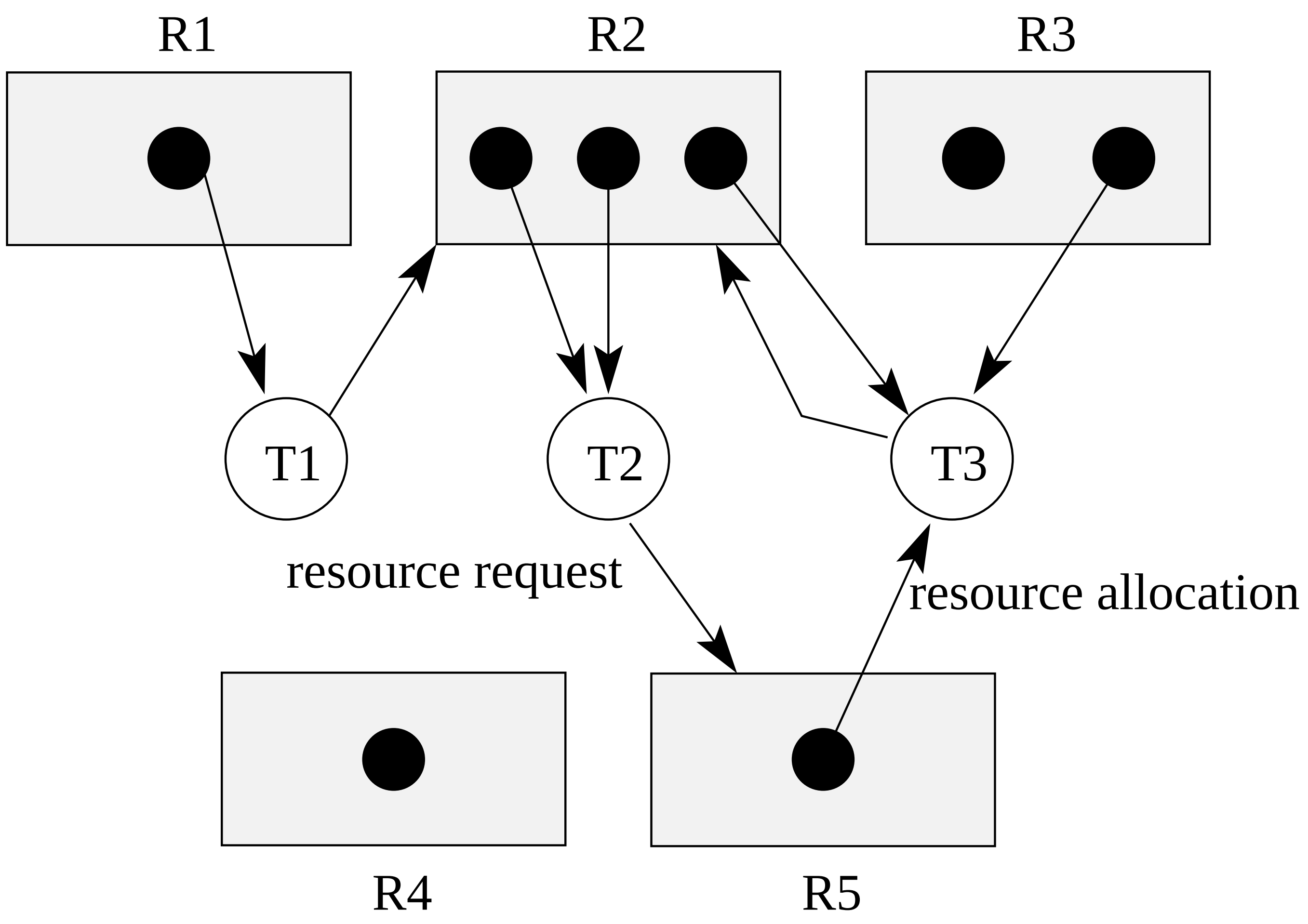
These two threads are *deadlocked* - neither thread can make progress. Waiting will not resolve the deadlock. The threads are permanently stuck.

Deadlocks (Another Simple Example)

- Suppose a machine has 64 MB of memory. The following sequence of events occurs.
  1. Process A starts, using 30 MB of memory.
  2. Process B starts, also using 30 MB of memory.
  3. Process A requests an additional 8 MB of memory. The kernel blocks process A's thread, since there is only 4 MB of available memory.
  4. Process B requests an additional 5 MB of memory. The kernel blocks process B's thread, since there is not enough memory available.

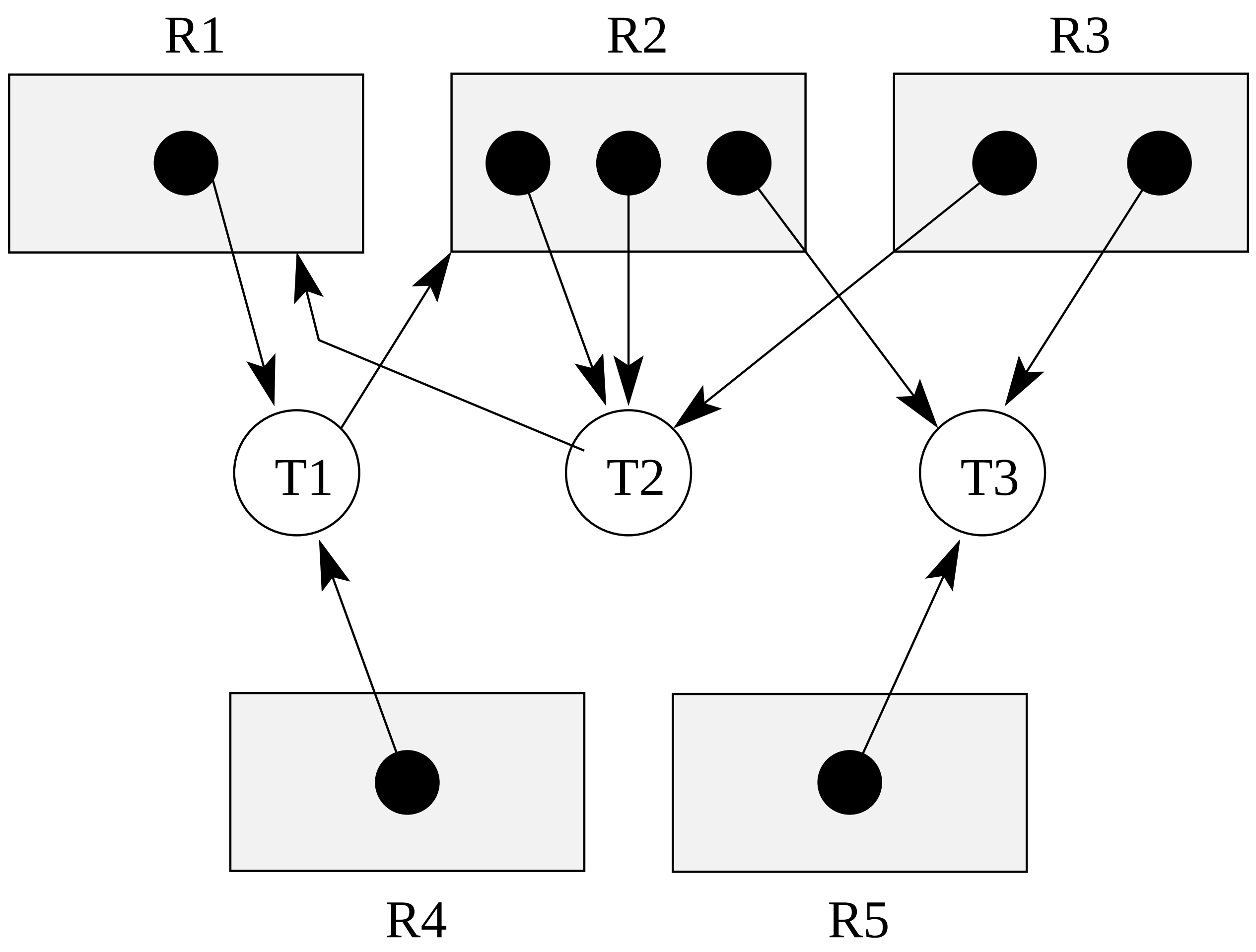
These two processes are deadlocked.

Resource Allocation Graph (Example)



Is there a deadlock in this system?

Resource Allocation Graph (Another Example)



Is there a deadlock in this system?

Deadlock Prevention

No Hold and Wait: prevent a process from requesting resources if it currently has resources allocated to it. A process may hold several resources, but to do so it must make a single request for all of them.

Preemption: take resources away from a process and give them to another (usually not possible). Process is restarted when it can acquire all the resources it needs.

Resource Ordering: Order (e.g., number) the resource types, and require that each process acquire resources in increasing resource type order. That is, a process may make no requests for resources of type less than or equal to i if it is holding resources of type i.

Deadlock Detection and Recovery

- main idea: the system maintains the resource allocation graph and tests it to determine whether there is a deadlock. If there is, the system must recover from the deadlock situation.
- deadlock recovery is usually accomplished by terminating one or more of the processes involved in the deadlock
- when to test for deadlocks? Can test on every blocked resource request, or can simply test periodically. Deadlocks persist, so periodic detection will not "miss" them.

Deadlock detection and deadlock recovery are both costly. This approach makes sense only if deadlocks are expected to be infrequent.

Detecting Deadlock in a Resource Allocation Graph

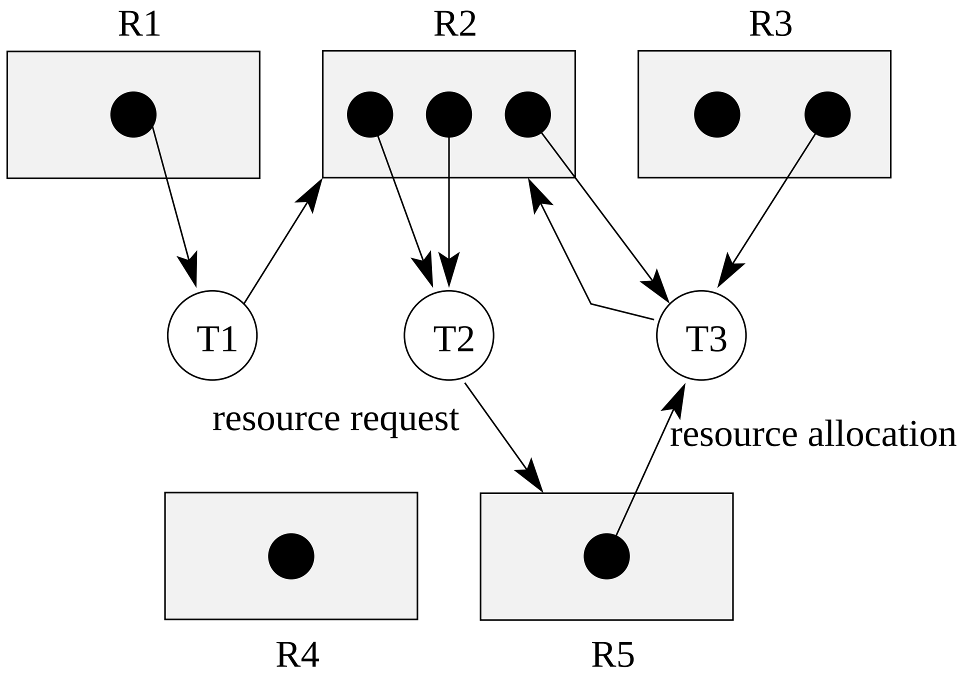
- System State Notation:
  - D_i: demand vector for process P_i
  - A_i: current allocation vector for process P_i
  - U: unallocated (available) resource vector
- Additional Algorithm Notation:
  - T: scratch resource vector
  - f_i: algorithm is finished with process P_i? (boolean)

Detecting Deadlock (cont'd)

```
/* initialization */
$T = U$
$f_i$ is false if $A_i > 0$, else true
/* can each process finish? */
while $\exists\ i\ (\ \neg\ f_i\ \wedge\ (D_i \leq T)\ )$ {
    $T = T + A_i$
    $f_i =$ true
}
/* if not, there is a deadlock */
if $\exists\ i\ (\ \neg\ f_i\ )$ then report deadlock
else report no deadlock
```

Deadlock Detection, Positive Example

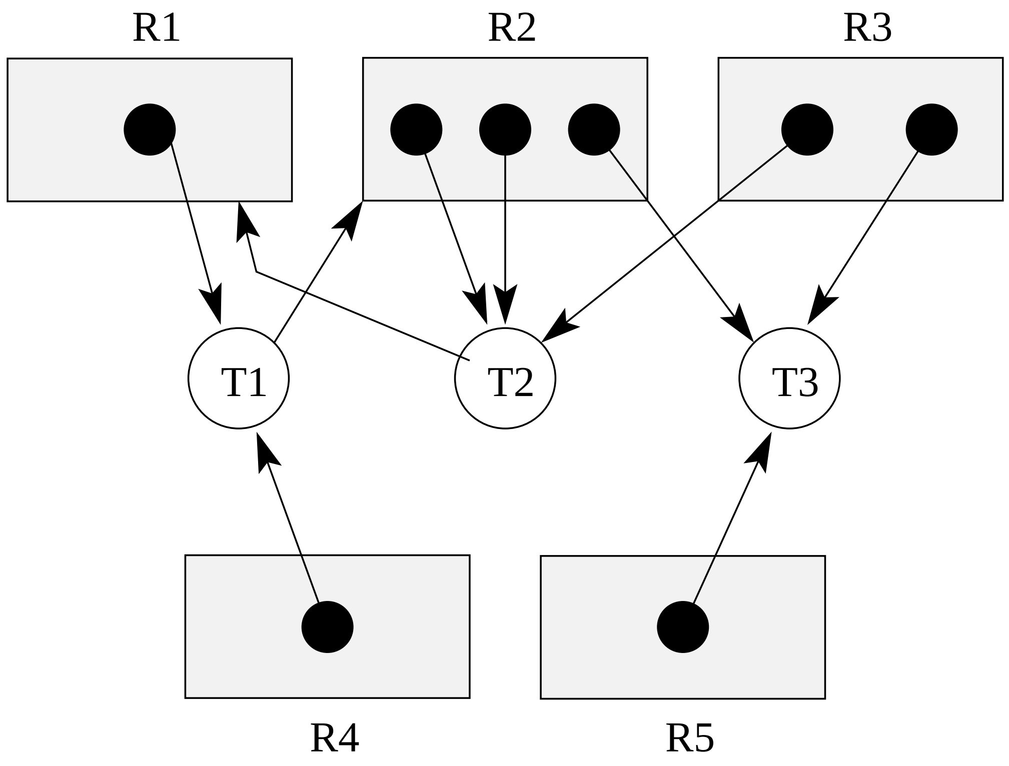
- $D_1 = (0, 1, 0, 0, 0)$
- $D_2 = (0, 0, 0, 0, 1)$
- $D_3 = (0, 1, 0, 0, 0)$
- $A_1 = (1, 0, 0, 0, 0)$
- $A_2 = (0, 2, 0, 0, 0)$
- $A_3 = (0, 1, 1, 0, 1)$
- $U = (0, 0, 1, 1, 0)$



The deadlock detection algorithm will terminate with $f_1 == f_2 == f_3 ==$ `false`, so this system is deadlocked.

Deadlock Detection, Negative Example

- $D_1 = (0, 1, 0, 0, 0)$
- $D_2 = (1, 0, 0, 0, 0)$
- $D_3 = (0, 0, 0, 0, 0)$
- $A_1 = (1, 0, 0, 1, 0)$
- $A_2 = (0, 2, 1, 0, 0)$
- $A_3 = (0, 1, 1, 0, 1)$
- $U = (0, 0, 0, 0, 0)$



This system is not in deadlock. It is possible that the processes will run to completion in the order P_3, P_1, P_2.