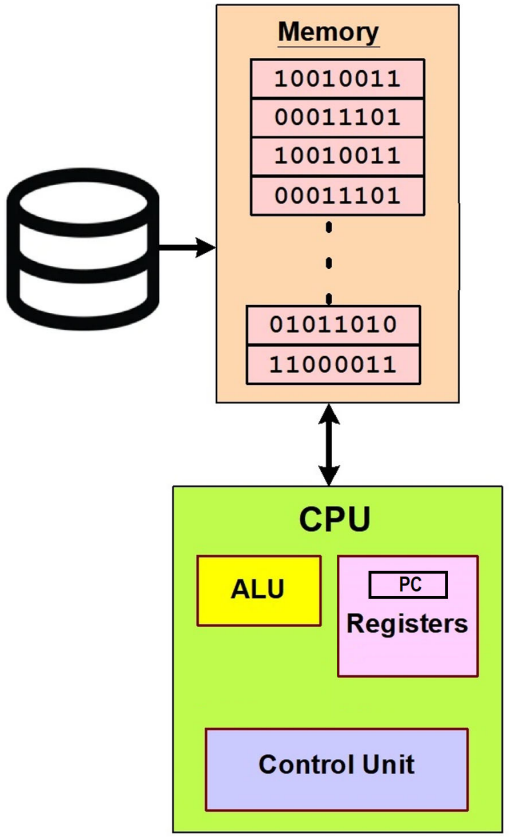
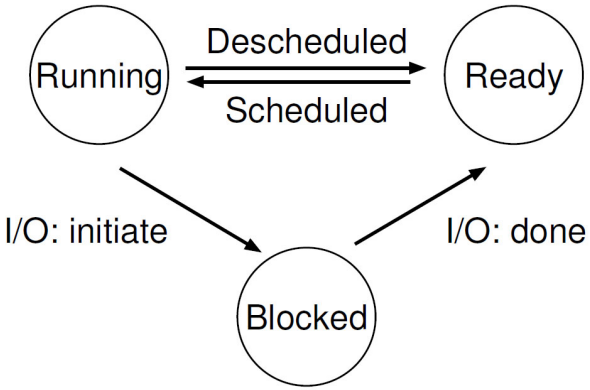
# EECE7376: Operating Systems Interface and Implementation

Concurrency

# Recap: Processes and Their States

- The **OS loader** creates a process for the program by loading it from the disk to the memory address space and mark it as "ready".
- When the **OS Scheduler** moves the process to the "running" state, the processor **program counter (PC)** is updated to point to the address of process's first instruction.
- A process can be in one of three states:
  - **Running** : The processor is executing the process instructions.
  - **Ready** (xv6's **Runnable**): In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
  - **Blocked** (xv6's **Sleep**): In the blocked state, a process has performed some kind of operation (e.g., system calls) that makes it not ready to run.



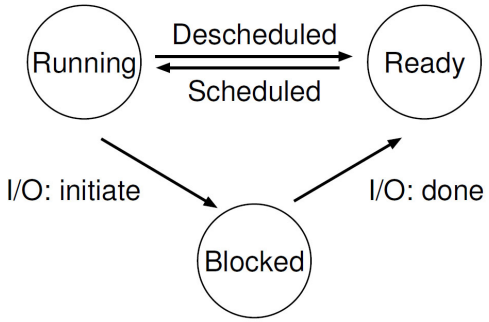




# Multi-Threaded Program

- Instead of our classic view of a single point of execution within a program (i.e., a single program counter, PC, where instructions are being fetched from and executed), a **multi-threaded program** has more than one point of execution, each of which is being fetched and executed from.
- Another way to think of this is that each **thread** is very much like a separate **process**, with the following **exceptions**:
  - Threads share the **same address space** and thus can access the same data.
  - Crash in one thread will **crash the entire process**.


# Why Use Threads?

- **Parallelism**: It is the task of transforming your standard single-threaded program into a program that does this sort of work by multiple threads to make programs run faster on modern hardware (e.g., multi core processors).
- **Overlapping**: Using threads is a natural way to avoid getting stuck; while one thread in your program waits (i.e., is blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful. Threading enables **overlap** of I/O with other activities within a single program.
- Using **multiple processes** to achieve the above features lacks the advantage that threads **share an address** space and thus make it easy to share data.




# Simple Thread Creation Code

- Download the `t0.c` code from:

https://github.com/remzi-arpacidusseau/ostep-code/tree/master/threads-intro
https://github.com/remzi-arpacidusseau/ostep-code/tree/master/include

- Compile using `gcc -pthread t0.c -o t0` and then run it.
- *Observation*:
  - The code creates two threads, **`p1`** and **`p2`**, each of which runs the function **`mythread()`** to print "**`A`**" and "**`B`**" independently.
  - After creating the two threads, **`p1`** and **`p2`**, the main thread calls **`Pthread_join()`**, which waits for a particular thread to complete.
  - Overall, **three threads** were employed during this run: the **`main`** thread, **`p1`** thread, and **`p2`** thread.
  - We could even see "**`B`**" printed before "**`A`**"!
- *Conclusion*:
  - Threads are not like **regular function calls** as the scheduler might decide to run **`p2`** first even though **`p1`** was created earlier; there is no reason to assume that a thread that is created first will run first.

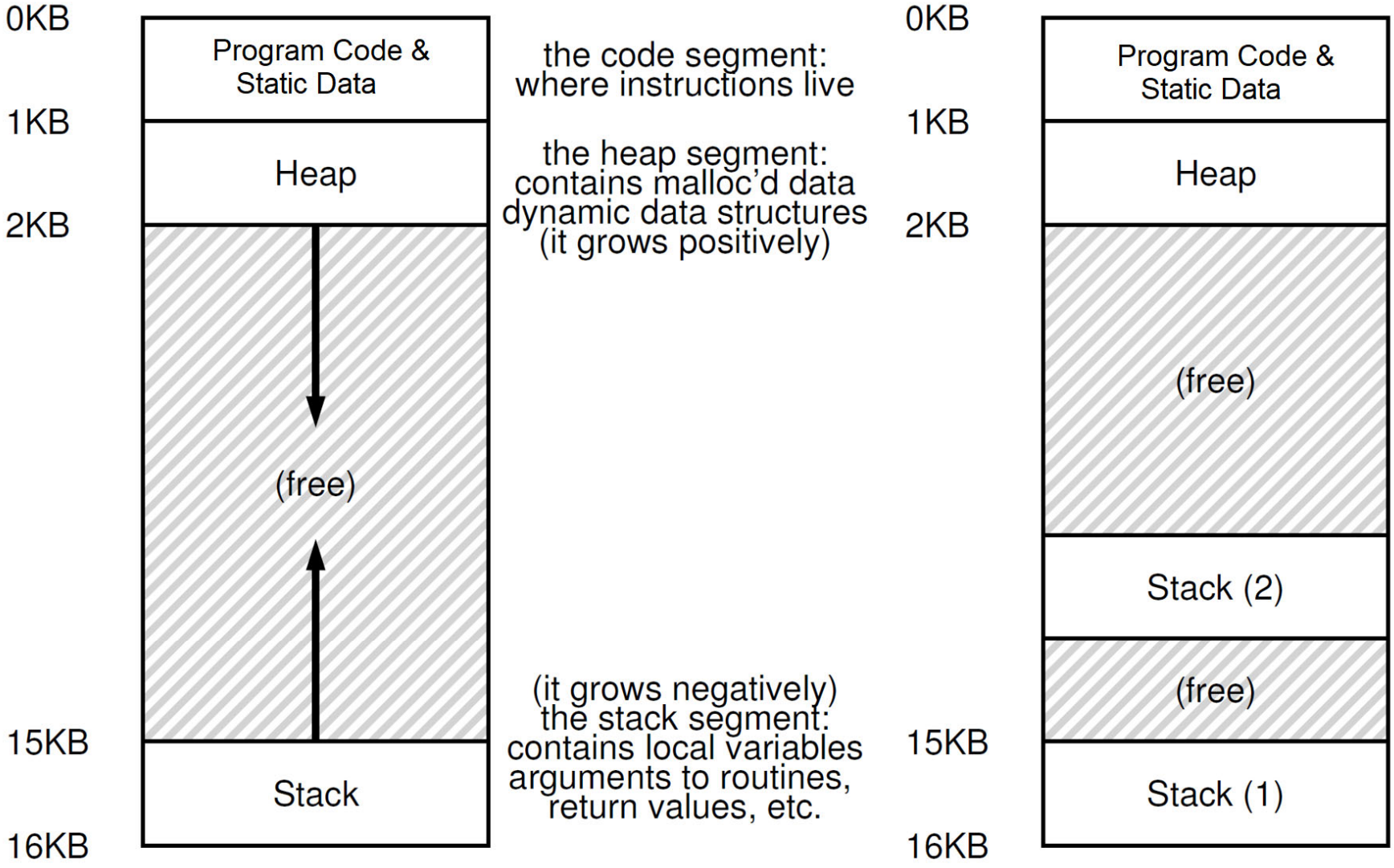
# t0.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg)      {
 printf("%s\n", (char *) arg);
    return NULL;               }

int main(int argc, char *argv[]) {
    if (argc != 1) {
        fprintf(stderr, "usage: main\n");
        exit(1);    }

    pthread_t p1, p2;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```


# Threads Context Switch

- When switching from running one thread T1 to running the other thread T2, a context switch must take place. The **register state** of T1 must be saved and the register state of T2 restored before running T2.
- The context switch between threads is quite similar to the context switch between processes, with the following **differences**:
  - With a process, its current state (e.g., registers values) are saved in a structure called **process control block** (**PCB**); now, we'll need one or more **thread control blocks** (**TCBs**).
  - The **address space remains the same** (i.e., there is no need to switch the **page table** we are using).
  - Instead of a single **stack** in the address space, there will be one per thread as each thread may call into various routines independently.

# Multi-Threaded Address Spaces




**Single-Threaded And Multi-Threaded Address Spaces**


# Threads With Shared Data

- Download the **`t1.c`** code , compile, and run it.

https://github.com/remzi-arpacidusseau/ostep-code/tree/master/threads-intro

- *Observation*:
  - Two threads are trying to increment by 1 the shared variable **`counter`** in a loop of a max iterations.
  - Instead of always receiving the desired final result of **`counter`** as 2 × max, we receive different result every time we run the program!
- *Conclusion*:
  - Adding 1 to **`counter`** is not one instruction executed by the processor, rather, it is a sequence of instructions like the following in RISC-V:

```
lw  x9, 4(x6)      //load counter from its address in the memory
add x9, x9, 1      //add 1
sw  x9, 4(x6)      //store back the updated value
```

  - If a thread gets context switch before "storing back" the updated **`counter`**, the other thread will get the same value of the **`counter`** before the update. This will result in both threads store back the same result even after performing two add instructions

# t1.c (1 of 2)

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

int max;
volatile int counter = 0; // shared global variable

void *mythread(void *arg) {
    char *letter = arg;
    int i; // stack (private per thread)
    printf("%s: begin [addr of i: %p]\n", letter, &i);
    for (i = 0; i < max; i++)
            counter = counter + 1; // shared: only one
    printf("%s: done\n", letter);
    return NULL;
}
```

# t1.c (2 of 2)

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: main-first <loopcount>\n");
        exit(1);    }

    max = atoi(argv[1]);
    pthread_t p1, p2;
    printf("main: begin [counter = %d] [%x]\n", counter,
           (unsigned int) &counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [counter: %d]\n [should: %d]\n",
           counter, max*2);
    return 0;
}
```


# Race Condition and Critical Section

- What we have demonstrated in the previous example is called a **race condition** (or **data race**) where the results depend on the timing execution of the code.
  - Instead of a nice **deterministic** computation, which we are used to from computers, we call this result **indeterminate**.
- Because multiple threads executing the **counter increment code** can result in a race condition, we call this code a **critical section**.
  - A **critical section** is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.
  - A **race condition** arises if multiple threads enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.


# Tools to Detect Race Condition

- **Valgrind** is a system for debugging and profiling Linux programs. With its tool suite you can automatically detect many memory management and threading bug. Resources:
  - https://valgrind.org/docs/manual/quick-start.html
  - https://valgrind.org/docs/manual/drd-manual.html
- Install valgrind by entering the following commands in the terminal:

```
$sudo apt update
$sudo apt install valgrind
```

- Make sure to compile your program with the `-g` option to include debugging information that allows the source code line numbers to be included in the valgrind report.

```
$gcc -g -pthread t1.c -o t1
```

- Using the executable file `t1` along with its argument, run the following command to let the tool check for race conditions:

```
$valgrind --tool=helgrind ./t1 100000
```


# Mutual Exclusion and Atomicity

- To solve the race condition, we want for the critical sections what we call **mutual exclusion**.
  - Mutual exclusion guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.
- Atomicity, in this context, means "**as a unit**", which sometimes we take as "**all or none**."
- What we'd like is to execute the three-instruction sequence atomically:

```
lw  x9, 4(x6)
add x9, x9, 1
sw  x9, 4(x6)
```

- How can we do that with the presence of interrupts?


# Locks: The Basic Idea

- Locks can be used to provide mutual exclusion on code critical sections as in the following example:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
 ...
Pthread_mutex_lock(&mutex);
 counter = counter + 1;
Pthread_mutex_unlock(&mutex);
```

- A **lock** is just a variable, such as `mutex` above.
  - Programmers should assign a **dedicated lock** variable to **every critical section** (this will help in increasing concurrency among threads).
- This lock holds the state of the lock at any instant in time.
- It is either **available** (**unlocked**, **free**) or **acquired** (**locked**, **held**).
- At anytime, exactly one thread holds the lock and presumably the thread is in a critical section.


# `lock()` and `unlock()` Routines

- Calling the routine `lock()` tries to acquire the lock; if no other thread holds the lock (i.e., it is free), the thread will acquire the lock and enter the critical section; this thread is sometimes said to be the **owner** of the lock.
- If another thread then calls `lock()` on that same lock variable (`mutex` in our example), it **will not return while the lock is held** by another thread; in this way, other threads are prevented from entering the critical section while the first thread that holds the lock is in there.
- Once the owner of the lock calls `unlock()` and there are waiting threads (stuck in `lock()`), one of them will (eventually) **notice** (or **be informed of**) this change of the lock's state, acquire the lock, and enter the critical section.


# Criteria of Building a Lock

- An efficient implementation of a lock involves all three levels of a computing system:
  1. The hardware
  2. The OS
  3. The user-level library software
- Basic **criteria** to evaluate whether a lock works well:
  - **Mutual exclusion**: Does the lock prevent multiple threads from entering a critical section?
  - **Fairness**: Does any thread contending for the lock **starve** while doing so, thus never obtaining it?
  - **Performance**: What is the time overheads added by using the lock?


# Locking by Controlling Interrupts

- One way to build a lock is by **disabling interrupts** for critical sections (i.e., disable interrupts on lock and re-enable them back on unlock).
  - By turning off interrupts (using special hardware instruction) before entering a critical section, we ensure that the code inside the critical section will not be interrupted, and thus will execute as if it were atomic.
- The positive of this approach is its **simplicity**.
- The negatives are:
  - Trusting user programs to perform a *privileged* operation. Malicious program could call `lock()` and go into an **endless loop**. The OS never regains control of the system unless we restart it.
  - It does not work on **multiprocessors** as threads will be able to run on other processors.
  - turning off interrupts for extended periods of time can lead to **missing important interrupts** that leads to serious systems problems (e.g., the CPU misses the fact that a disk device has finished a read request).


# Locking by Test-and-Set: Hardware

- Here the hardware provides instructions that allow for **atomic exchange** (read/write) memory operation.
- In RISC-V, the following two instructions are provided:

  **Load Reserved**: `LR.W rd,(rs1)`

  **Store Conditional**: `SC.W rd,(rs1),rs2`
- The memory address in both instructions must be the same (usually it is the mutex address).
- `LR.W` loads to `rd` a word from the memory address in `rs1` and places a "**reservation**" on that memory address (one way is to store this address in a special register).
- `SC.W` attempts to store `rs2` content to the address in `rs1`.
  - Succeeds if a **valid reservation still** exists on the memory address, **invalidates** that reservation, stores `rs2` content in the memory address, and **returns 0** in `rd`.
  - Fails if location has **invalid reservation** and **returns non-zero** value in `rd`

# RISC-V Lock and Unlock Code

- **Lock**: Assume address of the **mutex** is passed in register **x20** and it is initialized to zero to represent an unlocked status.
  Note: in RISC-V register x0 always has the value zero

```
        addi x12,x0,1          // x12 has 1 as the locked value
again:  lr.w x10,(x20)         // Read mutex and place a "reservation" on it
        bne  x10,x0,again      // Keep trying until it is 0 (i.e., unlocked)
        sc.w x11,(x20),x12     // Attempt to store 1 to the mutex to lock it.
             // If mutex still has a valid reservation → Success and store zero in x11
        bne  x11,x0,again      // Try again if fails (i.e., x11 is not zero).
             // This happens if another thread defeated us!
```

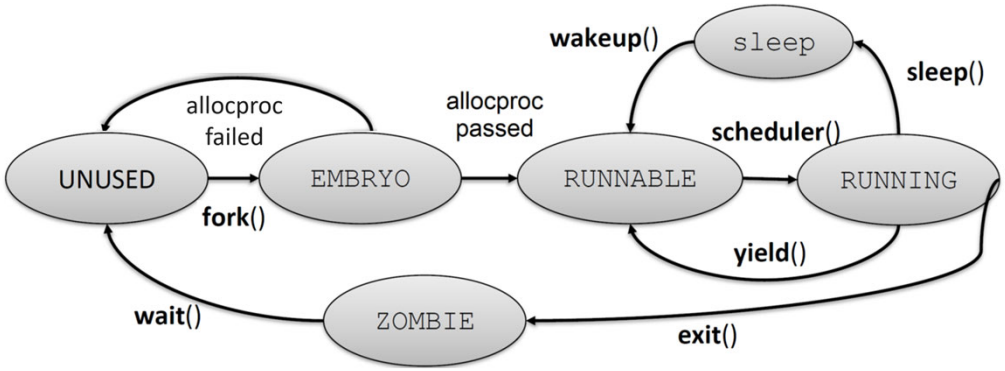
  - Note: `sc.w` cannot be replaced with a regular `sw` instruction, why?

- **Unlock**: Only used by the process that holds the lock

```
        sw   x0,0(x20)         // free lock
```

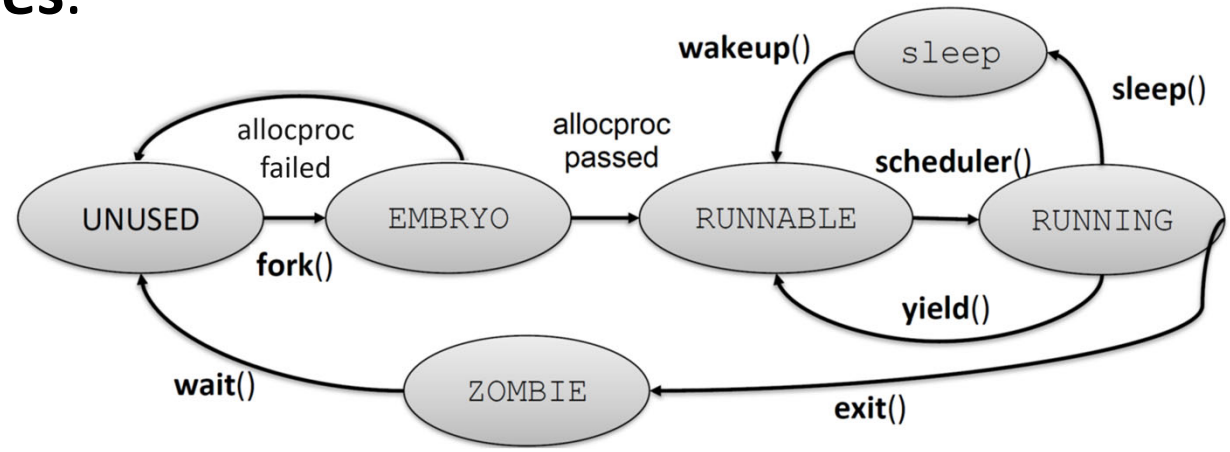
# Spin Lock

- The presented **Test-and-Set** lock is usually referred to as a **spin lock** as it simply spins, using CPU cycles, until the lock becomes available.
- To work correctly on a single processor, it requires a **preemptive scheduler** (i.e., one that will interrupt a thread via a timer, in order to run a different thread, from time to time).
- Without preemption, spin locks don't make much sense on a single CPU, as a thread spinning on a CPU will never surrender it.



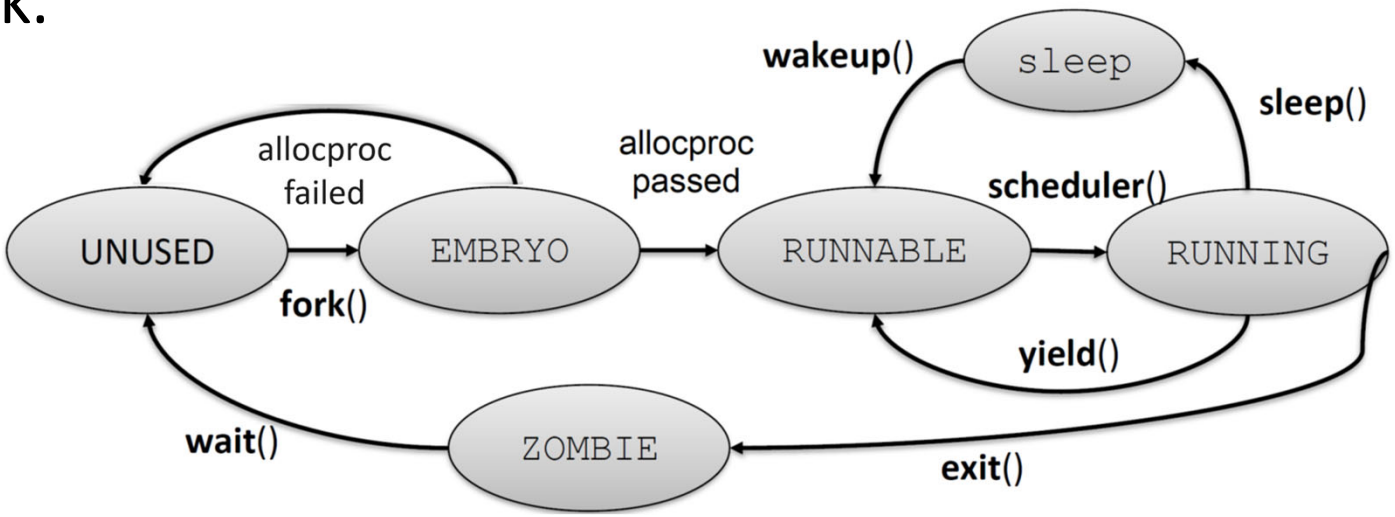
# Spin Lock Fairness and Performance

- The simple spin lock that has been discussed thus far is **not fair** and may lead to **starvation**.
- In the single CPU case, performance overheads can be quite painful.
  - Imagine the case where the thread holding the lock is preempted within a critical section. The scheduler might then run every other waiting thread. Those threads will spin for **the duration of a time slice** before giving up the CPU, **a waste of CPU cycles**.



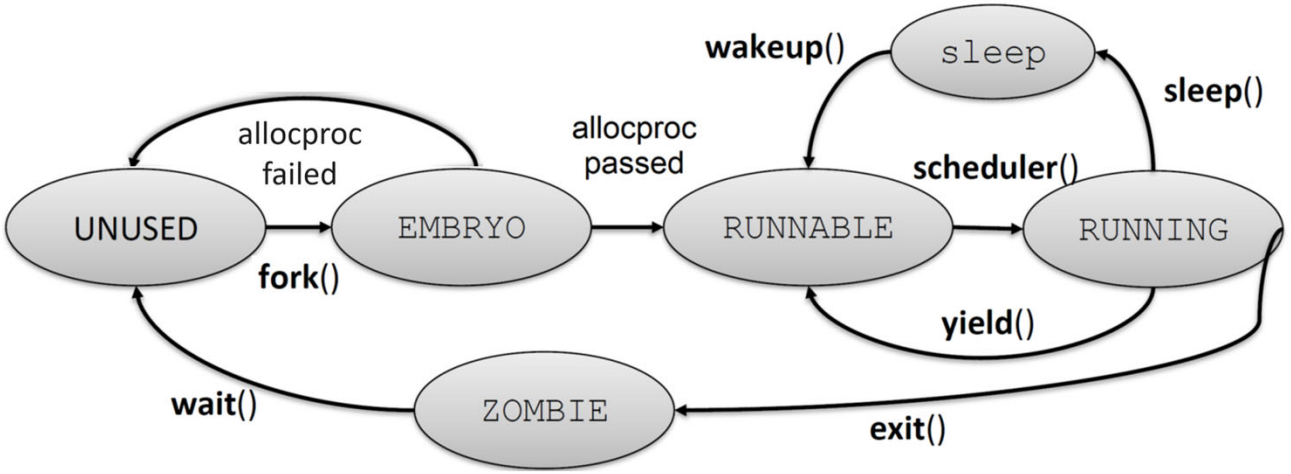
# Operating System Primitive `yield()` (1/2)

- Hardware supports us with a working lock. We still have a problem: what to do when a context switch occurs in a critical section, and threads start to **spin endlessly**, waiting for the interrupted (lock-holding) thread to be run again?
- One solution is for the OS to provide a primitive **`yield()`** which allows a thread to give up the CPU and move back to **Runnable** state when it finds out that another thread is holding the lock.



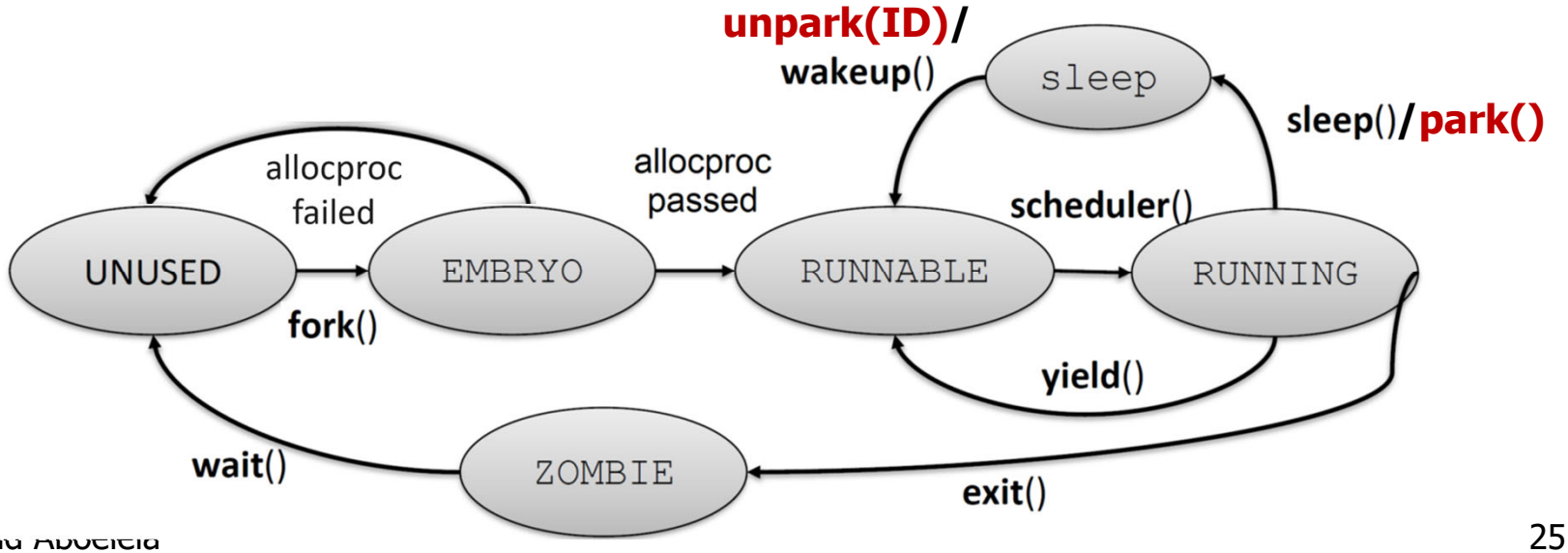
# Operating System Primitive `yield()` (2/2)

- Yield is simply a system call that moves the caller from the **Running** state to the **Ready** (xv6's **Runnable**) state right away (instead of spinning for the duration of a time slice), and thus promotes another thread to running. Thus, the **yielding** thread essentially **de-schedules** itself.
- While better than the spinning approach, this approach is still **costly**; the cost of a **context switch** can be substantial.
- Also, it does not tackle the **starvation** problem.



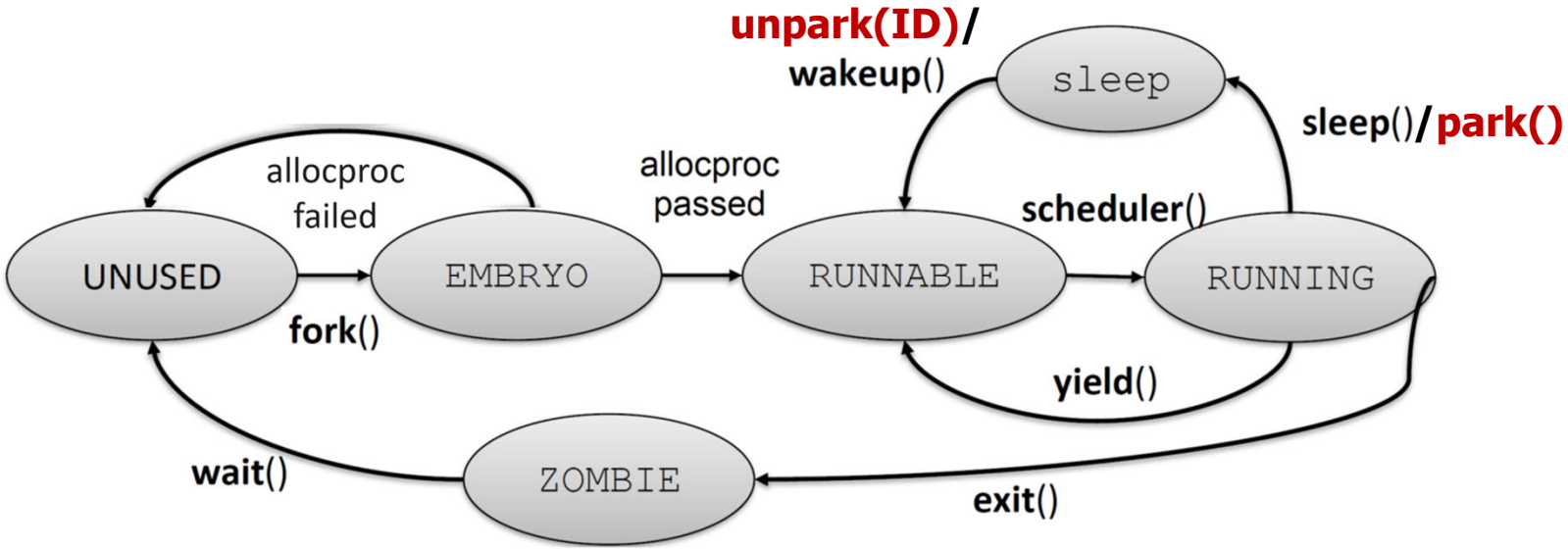
# Sleeping in a Queue (1/2)

- In the approaches we studied so far, the **scheduler** determines which thread runs next and it might make the following **bad choice**:
  - Run a thread that must either **spin** waiting for the lock (our first approach) or **yield** the CPU immediately (our second approach).
- The operating system *Solaris* provides two primitives:
  - **`park()`** to put a calling thread to **sleep** (not to ready like yield),
  - **`unpark(threadID)`** to **wake up** a particular thread.



# Sleeping in a Queue (2/2)

- With the park and unpark primitives, a **queue** of lock waiters (the threads that have been "parked") can be utilized to make a more efficient lock. The queue helps control who gets the lock ("unparked") next and thus **avoids starvation**.
  - When a thread calls lock and the lock is not available, it is added to the queue and parked.
  - When a thread calls unlock, a parked thread is unparked to the "test and set" operation to acquire the lock.




# Thread Unsafe vs Thread Safe Counter

Unsafe

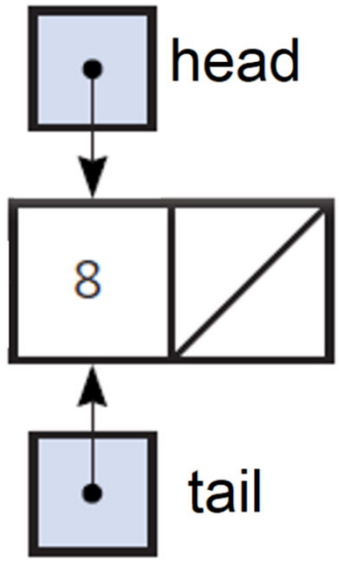
```
typedef struct __counter_t
{int value;} counter_t;

void init(counter_t *c)
{c->value = 0; }

void increment(counter_t *c)
{c->value++; }

void decrement(counter_t *c)
{c->value--; }

int get(counter_t *c)
{ return c->value; }
```

Safe

```
typedef struct __counter_t {
  int value;
  pthread_mutex_t lock;
 } counter_t;

void init(counter_t *c){//called before creating any threads
   c->value = 0;
   Pthread_mutex_init(&c->lock, NULL); }

void increment(counter_t *c) {
  Pthread_mutex_lock(&c->lock);
   c->value++;
  Pthread_mutex_unlock(&c->lock); }

void decrement(counter_t *c) {
  Pthread_mutex_lock(&c->lock);
   c->value--;
  Pthread_mutex_unlock(&c->lock); }

int get(counter_t *c) {
  Pthread_mutex_lock(&c->lock);
   int rc = c->value;
  Pthread_mutex_unlock(&c->lock);
 return rc; }
```

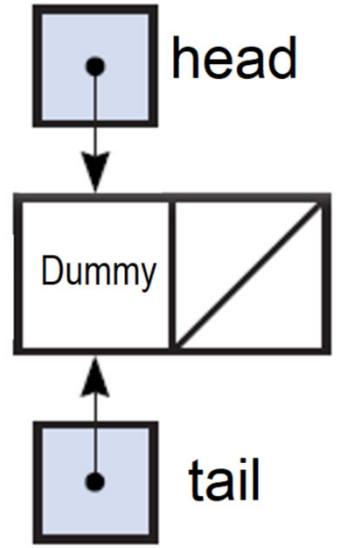

# Single vs. Multiple Locks

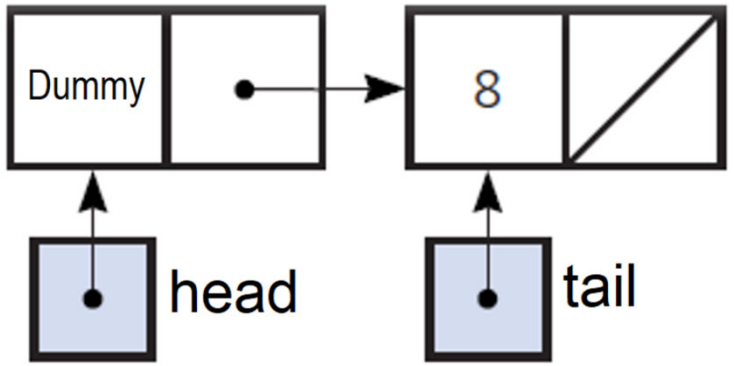
- Using a **single lock** with all critical sections has a performance problem.
- If your data structure is too slow, you'll have to do more than just adding a single lock.
- For example, with a queue data structure we can use two locks to enable concurrency of **enqueue** and **dequeue** operations.
- However, dequeuing a one-node queue or enqueuing in an empty queue would require **updating both tail and head** !



# Michael & Scott Queue Algorithm

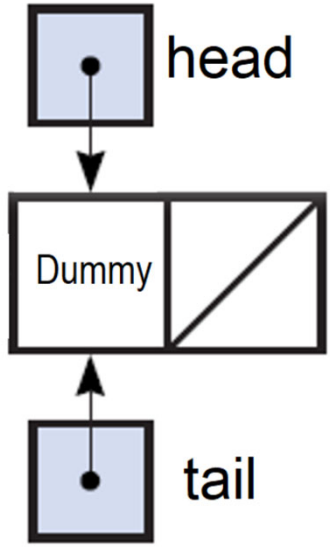
- One trick used by **Michael and Scott** is to add a **dummy node** (allocated in the queue initialization code); where the **head** always points to this dummy node.
  - The dummy node enables the **separation of head and tail operations** as dequeuing a one-node queue would require **updating only the head** and enqueuing in an empty queue would require **updating only the tail**.
- The code in the following pages implements a thread-safe queue data structure using two locks, one for the head of the queue, and one for the tail.



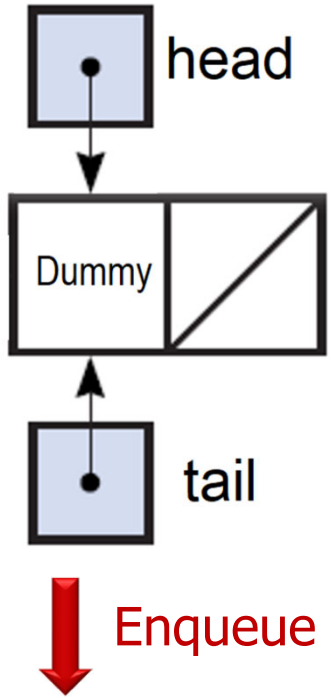
# Michael & Scott Concurrent Queue (1/3)

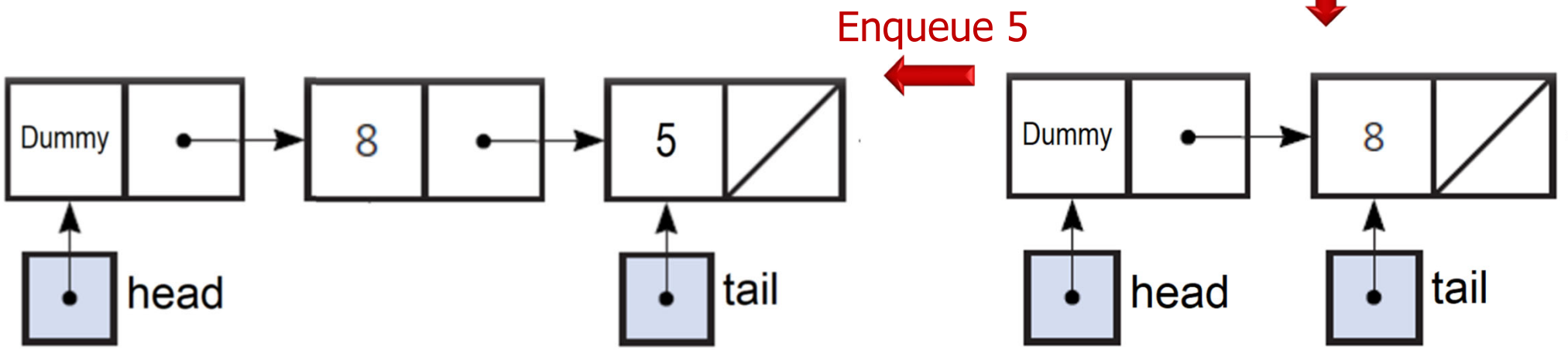
```
typedef struct __node_t {
  int value;
  struct __node_t *next;
} node_t;

typedef struct __queue_t {
  node_t *head;
  node_t *tail;
  pthread_mutex_t head_lock, tail_lock;
} queue_t;

void Queue_Init(queue_t *q) {
  //Creating the dummy node, which is needed so that "tail" update is separated from "head" update.
  node_t *tmp = malloc(sizeof(node_t));
  tmp->next = NULL;  //a dummy node has next=NULL and no need to set its value attribute
  q->head = q->tail = tmp;
  pthread_mutex_init(&q->head_lock, NULL);
  pthread_mutex_init(&q->tail_lock, NULL);
}
```



# Michael & Scott Concurrent Queue (2/3)

```
// Add a new node at the tail with content "value"
void Queue_Enqueue(queue_t *q, int value) {
 node_t *tmp = malloc(sizeof(node_t));
 assert(tmp != NULL);
 tmp->value = value;  tmp->next = NULL;
 pthread_mutex_lock(&q->tail_lock);
  q->tail->next = tmp;
  q->tail = tmp;// tail always points to the last enqueued node or
                // to the dummy node (if the queue is empty)
 pthread_mutex_unlock(&q->tail_lock);
}
```

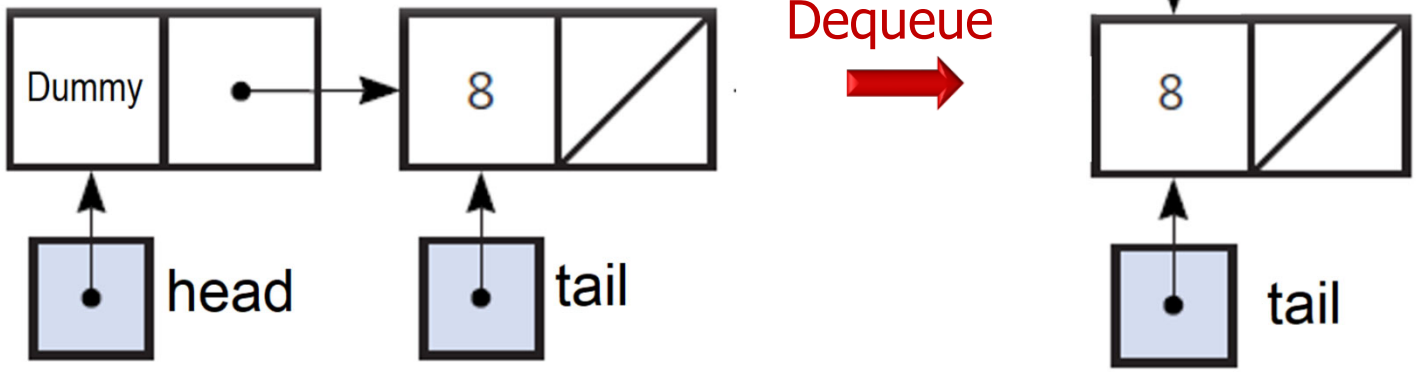


# Michael & Scott Concurrent Queue (3/3)

```
// Remove the first node in the queue (if it exists) and return its value in the "value" parameter.
// The function returns 0 on success and -1 on failure.
int Queue_Dequeue(queue_t *q, int *value) {
 pthread_mutex_lock(&q->head_lock);
  node_t *tmp = q->head;     node_t *new_head = tmp->next;
  if (new_head == NULL) {// indicates an empty queue with only a dummy node
      pthread_mutex_unlock(&q->head_lock);
      *value = -1;
      return -1;          }
  *value = new_head->value;
  q->head = new_head;  //head always points to a dummy node,
   //which is either the original dummy node or most recently dequeued node
 pthread_mutex_unlock(&q->head_lock);
 free(tmp);  // old dummy node is removed
 return 0; }
```

Why wasn't the node with value 8 explicitly removed?




# Concurrency with Conditions

- Locks are not the only primitives that are needed to build concurrent programs.
- There are many cases where a thread wishes to check whether a **condition** is true before continuing its execution.
- For example, a parent thread might wish to check whether a child thread has completed before continuing (this is often done by calling `join()`);
- How should such a wait be implemented?
- We could try using a shared variable like in the example:

```
while (done == 0) ; // spin
```

  - This solution generally works, but it is hugely inefficient as the parent **spins** and **wastes CPU time**.
  - Instead, we need a way to put the parent to **sleep** until the condition it is waiting for (e.g., the child is done executing) comes true.


# Condition Variables

- A **condition variable** is an explicit queue that threads can put themselves on when **waiting** on the condition.
- Some other thread, when it changes said condition, can then **wake** one (or more) of those waiting threads.
- To declare such a condition variable, one simply writes something like this: `pthread_cond_t c` to declare `c` as a condition variable
- A condition variable has two operations associated with it: `wait()` and `signal()`.
  - The `wait()` call is executed when a thread wishes to put itself to **sleep**;
  - The `signal()` call is executed when a thread has changed something in the condition and thus wants to **wake** a sleeping thread waiting on this condition (queue).

# Condition Variable Implementation (1/2)

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit() {
 Pthread_mutex_lock(&m);
 done = 1;
 Pthread_cond_signal(&c);
 Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
 printf("child\n");
 thr_exit();
 return NULL;
}

void thr_join() {
 Pthread_mutex_lock(&m);
 while (done == 0) //while, not if, so when wait returns, one more while iteration is used to confirm that done is 0
     Pthread_cond_wait(&c, &m); //wait() unlocks m
 Pthread_mutex_unlock(&m);
}
```

```
int main(int argc, char *argv[]) {
 printf("parent: begin\n");
 pthread_t p;
 Pthread_create(&p, NULL, child, NULL);
 thr_join();
 printf("parent: end\n");
 return 0;
}
```

Why is a lock needed?


# Condition Variable Implementation (2/2)

- A running scenario of the previous code is:
  - The parent creates the child thread but continues running itself and thus immediately calls into `thr_join()` to wait for the child thread to complete.
  - In this case, it will acquire the lock, check if the child is done (it is not), and put itself to sleep by calling `wait()` (hence releasing the lock).
  - The child will eventually run, print the message "child", and call `thr_exit()` to wake the parent thread; this code just grabs the lock, sets the state variable done, and signals the parent thus waking it.
  - Finally, the parent will run (returning from `wait()` with the lock held), unlock the lock, and print the final message "parent: end".
- Using the lock is important to avoid the race case when the child interrupts the parent right after the parent checks `done` but before it calls `wait()`. Then the child calls `thr_exit()` and signals while there is no thread waiting and hence no thread is woken. Later when the parent runs `wait()`, it will sleep forever.

  What if the child interrupts the parent before the parent calls lock?


# Producer/Consumer Problem (1 of 2)

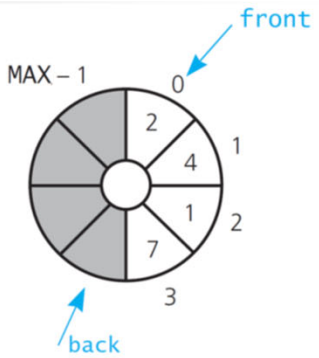
- The producer/consumer problem is another synchronization problem that involves one or more **producer threads** and one or more **consumer threads**.
- Producers generate data items and place them in a buffer; consumers grab said items from the buffer and consume them in some way.
  - *Example:* in a multi-threaded web server where producers put HTTP requests into a work **queue** (**a bounded buffer**) and consumers take requests out of this queue and process them.


# Producer/Consumer Problem (2 of 2)

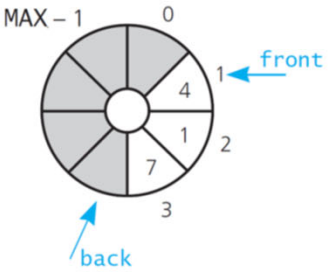
- Because the bounded buffer is a shared resource, we must require synchronized access to it.
- We will solve the Producer/Consumer problem using a simple queue buffer of integers.
  - The **producers** put integers in the buffer unless it is already **filled**.
  - The **consumers** get integers from the buffer unless it is **empty**.
- As speed is critical in such applications, an array is used to implement the bounded buffer queue.
  - Recall in the Michael & Scott concurrent queue implementation we used linked list dynamically created in the heap and hence there was no need to check for a filled queue.
- How can we perform enqueue and dequeue operations with a queue implemented using a fixed size array?

# The Put and Get Routines

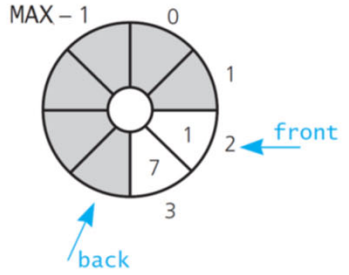
```
int buffer[MAX];  //implemented as a circular queue
int back  = 0;
int front = 0;
int count = 0;

//check if buffer is not full before calling put
void put(int value) {
 buffer[back] = value;
 back = (back + 1) % MAX;
 count++;
}

//check if buffer is not empty before calling get
int get() {
 int tmp = buffer[front];
 front = (front+ 1) % MAX;
 count--;
 return tmp;
}
```



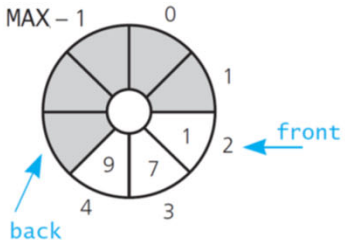

get()




get()




put(9)



# The Producer/Consumer Synchronization (1/2)

```
cond_t notfull, notempty;
mutex_t mutex;

void *producer(int loops) {
 int i;
 for (i = 0; i < loops; i++)    {
  Pthread_mutex_lock(&mutex);
  while (count == MAX) //buffer is full
         Pthread_cond_wait(&notfull, &mutex);
  put(i);                Pthread_cond_signal(&notempty);
  Pthread_mutex_unlock(&mutex); }
}

void *consumer(int loops) {
 int i, tmp;
 for (i = 0; i < loops; i++)  {
  Pthread_mutex_lock(&mutex);
  while (count == 0) //buffer is empty
         Pthread_cond_wait(&notempty, &mutex);
  tmp = get();           Pthread_cond_signal(&notfull);
  Pthread_mutex_unlock(&mutex);
  printf("%d\n", tmp);          }
}
```


# The Producer/Consumer Synchronization (2/2)

- The code uses **two condition variables** in order to properly signal which type of thread should wake up when the state of the buffer changes.
- The producer threads wait on the condition **notfull**, and signal **notempty**.
- The consumer threads wait on **notempty** and signal **notfull**.
- The code allows for **multiple values** to be produced before sleeping, and similarly multiple values to be consumed before sleeping. This reduces context switches.
- It also utilizes a **mutex** (lock) to allow for **concurrent producing or consuming** to take place.


# Semaphores

- So far, we utilized **locks** and **condition** variables to solve a broad range of relevant and interesting concurrency problems.
- **Dijkstra** and colleagues invented the **semaphore** as a single primitive for both locks and condition variables.
- A semaphore is an **object with an integer value** that we can manipulate with some routines.
- The semaphore object is defined as **a global object** that does not belong to any particular thread.
- **The initial value** of the semaphore determines its behavior.
- Therefore, before calling any routine to interact with the semaphore, we must first initialize it to a value appropriate to the needed behavior.


# POSIX Semaphores Routines

- The *Portable Operating System Interface* (POSIX) introduced the following semaphores routines:

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1); // initialize semaphore s to the value 1. The second
                    // argument is set to 0 to indicate that the semaphore is
                    // shared between threads in the same process.
sem_wait(&s);// decrement the value of semaphore s by one then block (wait in a
             // queue) if value of semaphore s is negative otherwise return right away.
sem_post(&s);// increment the value of semaphore s by one and wake one of the
             // waiting threads, if any. It returns right away.
// Note: the absolute value of the semaphore, when negative, equals to the number
// of waiting threads. This value generally isn't seen by users of the semaphores
```

# Semaphore Implementation (1/2)

- The following routines must be executed **atomically** so that no two processes can execute any of them on the same semaphore at the same time.
- This can be implemented using a similar approach to the one we studied to implement the mutex lock with assembly code.

```
sem_wait(&s){  //must be executed atomically (a critical section)
    s->value--;
    if (s->value < 0)
        block();  //calling process is placed in a waiting queue
  }

sem_post(&s){  //must be executed atomically (a critical section)
    s->value++;
    if (s->value >= 0)
        wakeup(P);  //remove a process from the waiting queue
}
```


# Semaphore Implementation (2/2)

- With each semaphore there is an associated waiting queue.
  - **Block** – place the process invoking the `sem_wait` routine on the appropriate waiting queue.
  - **Wakeup** – remove one of the processes in the waiting queue and place it in the ready queue.


# Semaphore as a Lock

- To use a semaphore as a lock, simply surround the critical section of interest with a `sem_wait()`/`sem_post()` pair.
- With the following code, only one thread can be running inside the critical section. Other thread will wait until the running thread calls `sem_post()`:

```
sem_t m;
sem_init(&m, 0, 1); // initialize to 1
…
sem_wait(&m);
  // critical section here
sem_post(&m);
```

- Because locks only have two states (held and not held), we sometimes call a semaphore used as a lock a **binary semaphore**.

# Semaphore as a Condition Variable

- This is an example code of using a semaphore as a condition variable where a *Parent* waits for its *Child* to complete its execution:

```
sem_t s;

void *child(void *arg) {
 printf("child\n");
 sem_post(&s); // signal here: child is done
 return NULL;
}

int main(int argc, char *argv[]) {
 sem_init(&s, 0, 0); // Initialized to 0 so that parent waits on sem_wait()
 printf("parent: begin\n");
 pthread_t c;
 Pthread_create(&c, NULL, child, NULL);
 sem_wait(&s); // wait here for child. What if child finishes before parent comes here?
 printf("parent: end\n");
 return 0;
}
```

# Semaphores for Producer/Consumer

```
sem_t mutex; //a lock implemented by semaphore
sem_t notfull;
sem_t notempty;

void *producer(int loops) {
 int i;
 for (i = 0; i < loops; i++) {
  sem_wait(&notfull);
  sem_wait(&mutex);
  put(i);
  sem_post(&mutex);
  sem_post(&notempty);        }
 }
```

- A mutex lock is needed in the code to avoid a race condition where multiple producers might try to update the same location in the buffer (with the problem of old data being overwritten).

```
void *consumer(int loops) {
 int i;
 for (i = 0; i < loops; i++) {
  sem_wait(&notempty);
  sem_wait(&mutex);
  int tmp = get();
  sem_post(&mutex);
  sem_post(&notfull);
  printf("%d\n", tmp);        }
}

int main(int argc, char *argv[])
{
  //Given the buffer has MAX capacity
 sem_init(&mutex, 0, 1);
 sem_init(&notfull, 0, MAX);
 sem_init(&notempty, 0, 0);
 // ...
}
```


# Semaphores for Thread Throttling (1/2)

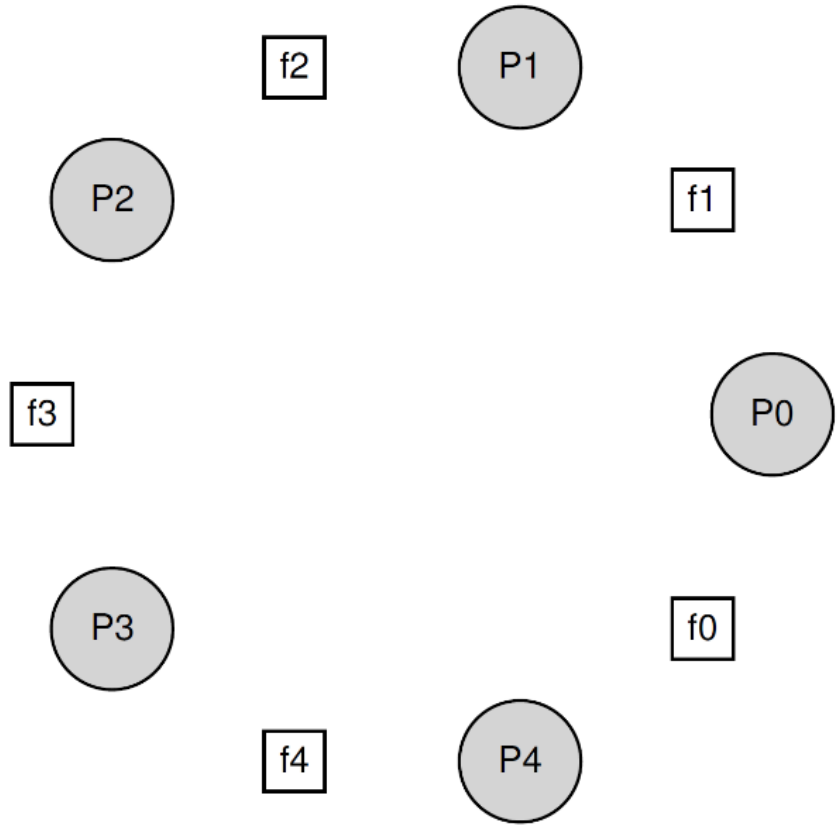
- **Thread throttling** is the approach of limiting the number of threads concurrently executing a piece of code.
- *Example:*
  - Assume hundreds of threads need to work on some problem in parallel. However, in a certain part of the code, each thread needs to acquire a large amount of memory.
  - If all threads enter the **memory-intensive** region at the same time, the sum of all the memory allocation requests will exceed the amount of physical memory on the machine. As a result, the machine will start thrashing (i.e., swapping pages to and from the disk), and the entire computation will slow to a crawl.


# Semaphores for Thread Throttling (2/2)

- A simple semaphore can solve this problem by initializing the value of the semaphore to the **maximum number of threads** allowed to enter the memory-intensive region at once, and then putting a `sem_wait()` and `sem_post()` around the region, a semaphore can naturally throttle the number of threads that are ever concurrently in the dangerous region of the code.

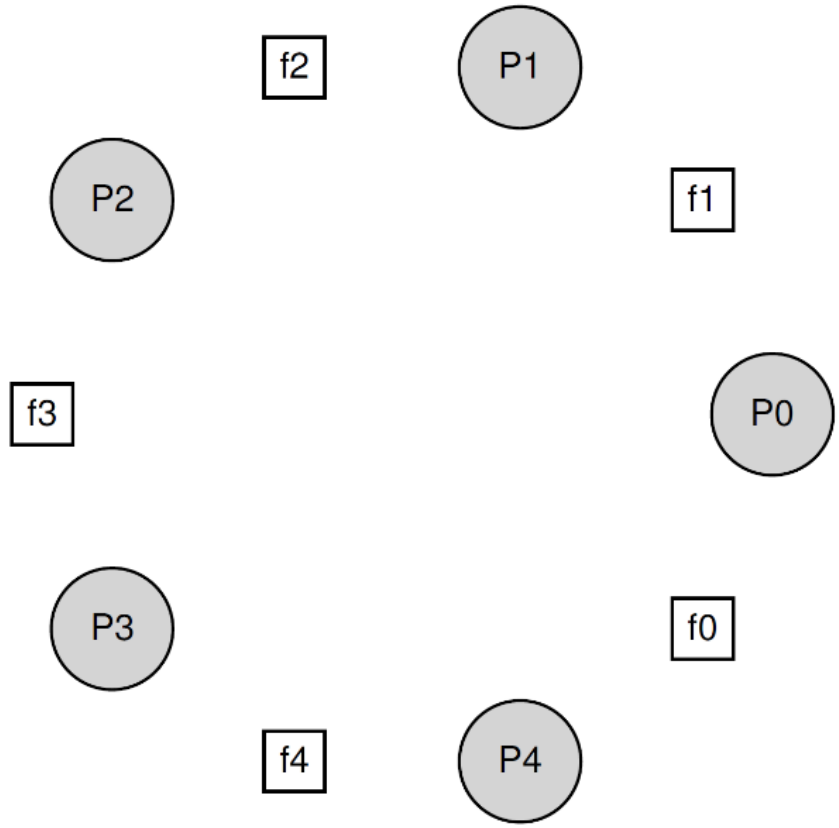
# The Dining Philosophers

- One of the most famous concurrency problems posed, and solved, by Dijkstra, is known as the **dining philosopher's** problem.
- Assume there are five "philosophers" sitting around a table. Between each pair of philosophers is a single fork.
- The philosophers each have times where they think, and don't need any forks, and times where they eat. **In order to eat, a philosopher needs two forks**, both the one on their left and the one on their right.
- The contention for these forks, and the synchronization problems that ensue, are what makes this a problem we study in concurrent programming.



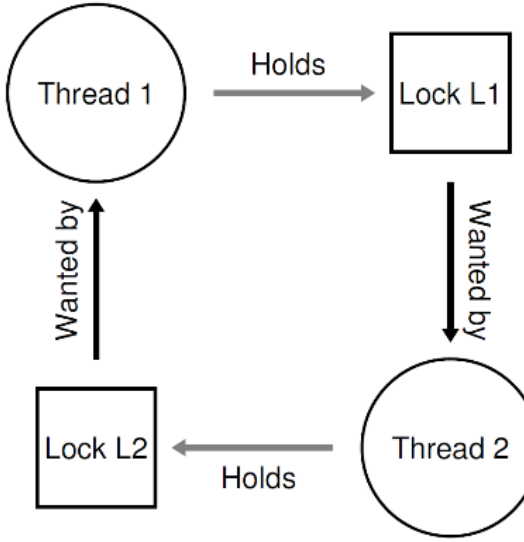
# The Dining Philosophers Deadlock

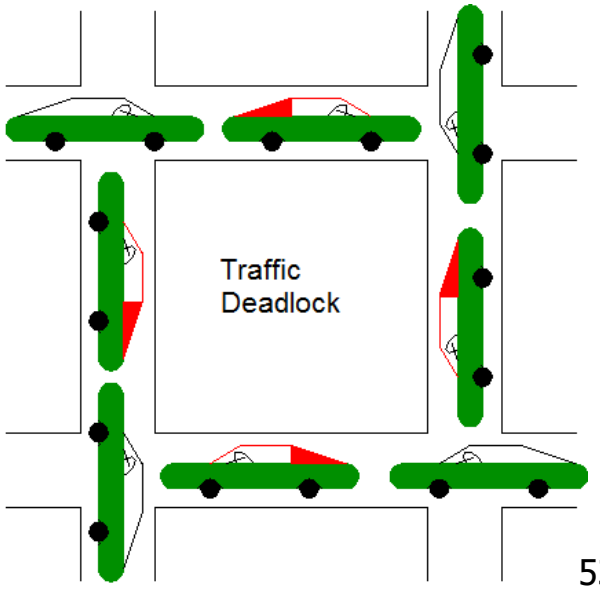
- If each philosopher happens to grab the fork on their left before any philosopher can grab the fork on their right, each will be stuck holding one fork and waiting for another, forever (**deadlock**).
- Dijkstra solved this deadlock problem by changing how forks are acquired by at least one of the philosophers.
- Assume that P4 tries to get the forks in a different order than the others (i.e., trying to grab right before left).
- This will allow P3 to grab the two needed forks to finish eating/releasing the forks and hence the cycle of waiting is broken.



# Deadlock Bugs

- Deadlock occurs, as shown in the example, when **Thread 1** is holding a lock **L1** and waiting for lock **L2**; and **Thread 2** that holds lock **L2** is waiting for **L1** to be released.
- The presence of a **cycle** in the graph is indicative of the deadlock.
- Simple deadlocks, as in this example, seem readily avoidable. For example, if Threads 1 and 2 both made sure to grab locks **in the same order**, the deadlock would never arise.
- However, the design of locking strategies in large systems must be carefully done to avoid deadlock in the case of circular dependencies that may occur naturally in the code



# Code with Potential Deadlock

```
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;

void* worker(void* arg) {
    if ((long) arg == 0) {
        Pthread_mutex_lock(&m1);
        Pthread_mutex_lock(&m2);
    } else {
        Pthread_mutex_lock(&m2);
        Pthread_mutex_lock(&m1);
    }
    Pthread_mutex_unlock(&m1);
    Pthread_mutex_unlock(&m2);
    return NULL;
}
```

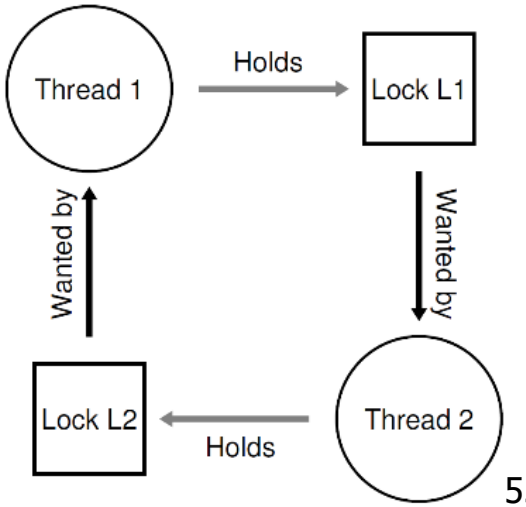
```
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    Pthread_create(&p1, NULL, worker,
                   (void *) (long) 0);
    Pthread_create(&p2, NULL, worker,
                   (void *) (long) 1);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    return 0;
}
```

*Note:* The **Valgrind** tool can be also used to detect potential deadlock in executable programs.

# Conditions for Deadlock

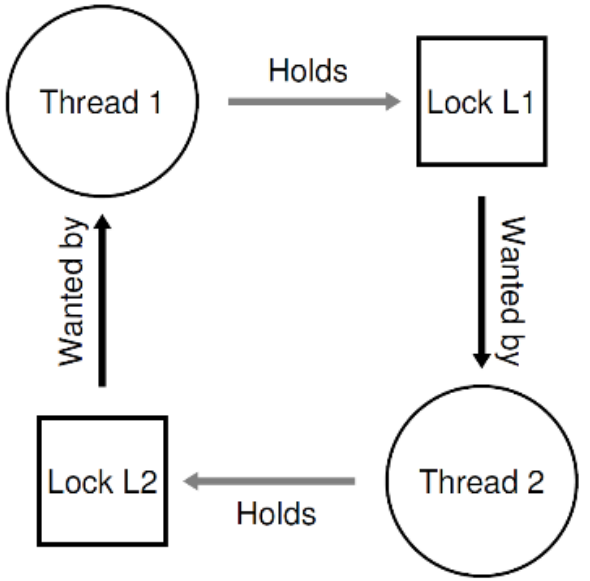
- Four conditions need to hold for a deadlock to occur:
  1) **Circular wait**: There exists a circular chain of threads such that each thread holds one or more resources (e.g., locks) that are being requested by the next thread in the chain.
  2) **No preemption**: Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
  3) **Hold-and-wait**: Threads hold resources allocated to them(e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
  4) **Mutual exclusion**: Threads claim exclusive control of resources that they require (e.g., only one thread at a time can acquire a specific lock).

- If any of these four conditions are not met, deadlock cannot occur.




# Preventing Circular Wait

- To prevent circular wait, write your locking code with a **total ordering** on lock acquisition.
  - For example, if there are only two locks in the system (L1 and L2), you can prevent deadlock by always acquiring L1 before L2. Such strict ordering ensures that no cyclical wait arises; hence, no deadlock
- Total lock ordering may be difficult to achieve and hence a **partial ordering** can be a useful way to structure lock acquisition so as to avoid deadlock.
- Lock ordering requires a deep understanding of the code base, and how various routines are called; just one mistake could result in a deadlock.



# Preventing No Preemption

- Many thread libraries provide the routine `pthread_mutex_trylock()` either grabs the lock (if it is available) and returns success or returns an error code indicating the lock is held.
- This routine can be used to avoid the **no preemption** problem:

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

- This approach **doesn't really add preemption** (the forcible action of taking a lock away from a thread that owns it), but rather uses the "try lock" approach to allow a developer to back out of lock ownership (i.e., **preempt their own ownership**) in a graceful way.

# Preventing Hold-and-wait

- The hold-and-wait requirement for deadlock can be avoided by **acquiring all locks at once**, atomically as follows:

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

- Here it requires that any time any thread grabs a lock, it first acquires the global **prevention** lock. This code guarantees that no untimely thread switch can occur in the midst of lock acquisition.
- This technique is likely to **decrease concurrency** as all locks must be acquired early on (at once) instead of when they are truly needed.


# Preventing Mutual Exclusion

- Avoiding mutual exclusion requires getting rid of the critical sections in the code.
- One way is to use powerful hardware instructions where data structures can be built in a manner that does not require explicit locking.
- As an example, the **atomic exchange** instruction sequence we discussed before. It can be used for example to atomically increment a value by a certain amount.

# Deadlock Avoidance via Scheduling

- Instead of deadlock prevention, in some scenarios **deadlock avoidance** is preferable.
- Avoidance requires the system to have some additional **priori information** of which locks various threads might grab during their execution, and subsequently schedules said threads in a way as to guarantee no deadlock can occur. This is generally is **not feasible**!
- *Example*: two processors run four threads (T1, T2, T3, T4). Each thread grabs (yes) or does not grab (no) locks L1 and L2 as shown below:

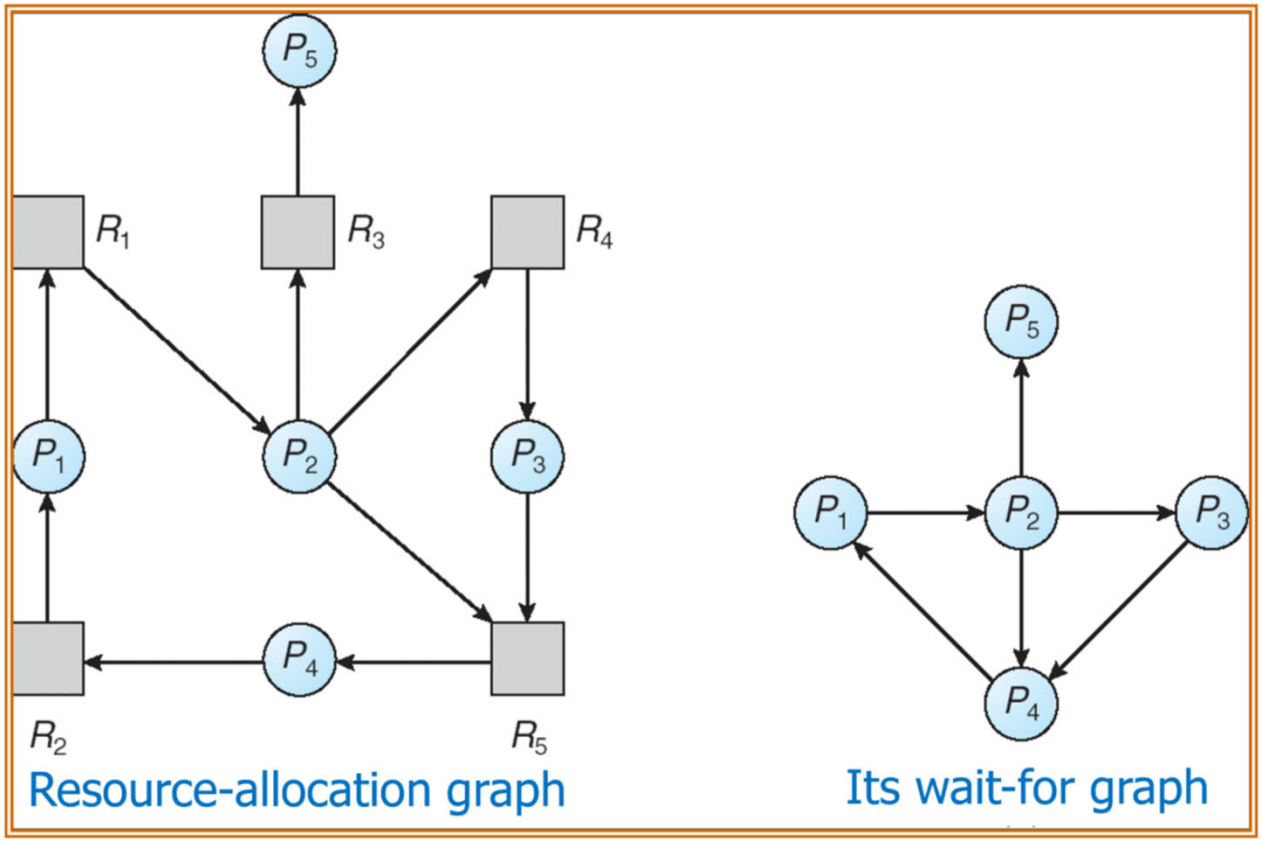
| | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| L1 | yes | yes | no | no |
| L2 | yes | yes | yes | no |

- A smart scheduler could thus compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise. Here is one such schedule:

# Deadlock Detect and Recover

- One strategy is to **allow deadlocks** to occasionally occur, and then take some action once such a deadlock has been **detected**.
- A deadlock detector runs periodically, building a processes "**wait-for**" graph from the **Resource-Allocation Graph (RAG)** and checking it for cycles.

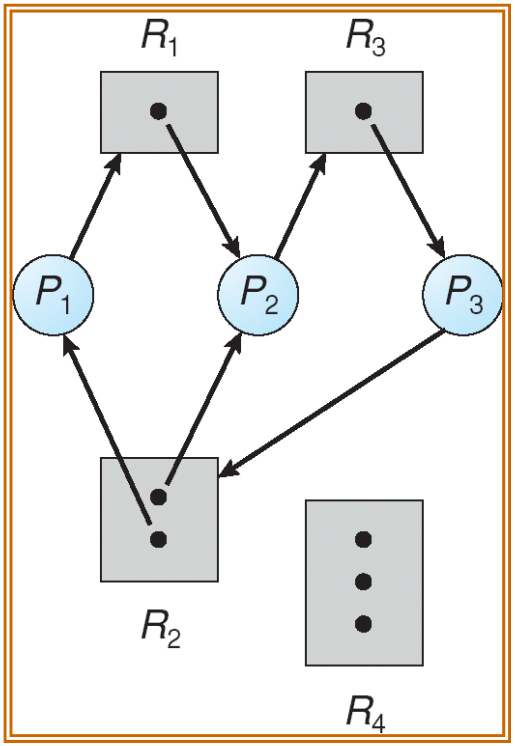
- To recover from a deadlock either abort all deadlocked processes or **abort one process at a time** until the deadlock cycle is eliminated.
- *Note:* if we didn't have a cycle (for example by removing $P_4$ waiting for $P_1$) then there would not be a deadlock. Why?



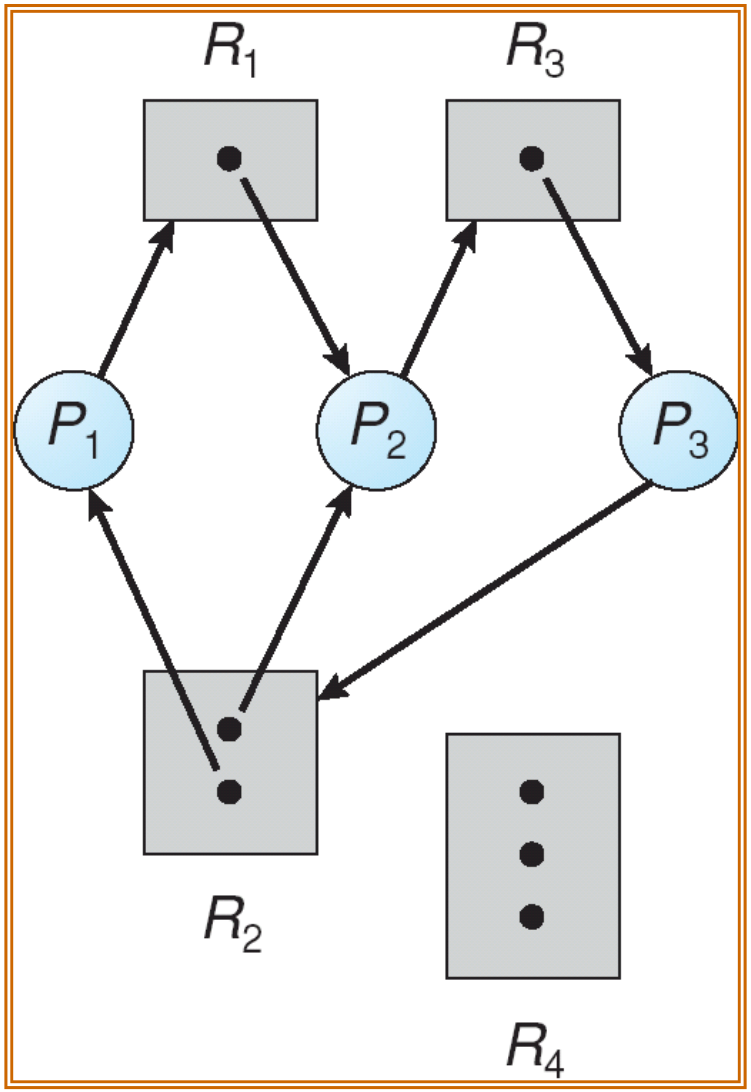
# Resource-Allocation Graph (RAG)

A set of vertices $V$ and a set of edges $E$.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$
- The **dots** inside the resource nodes represents the number of instances of the resource (e.g., $R_4$ has three instances)

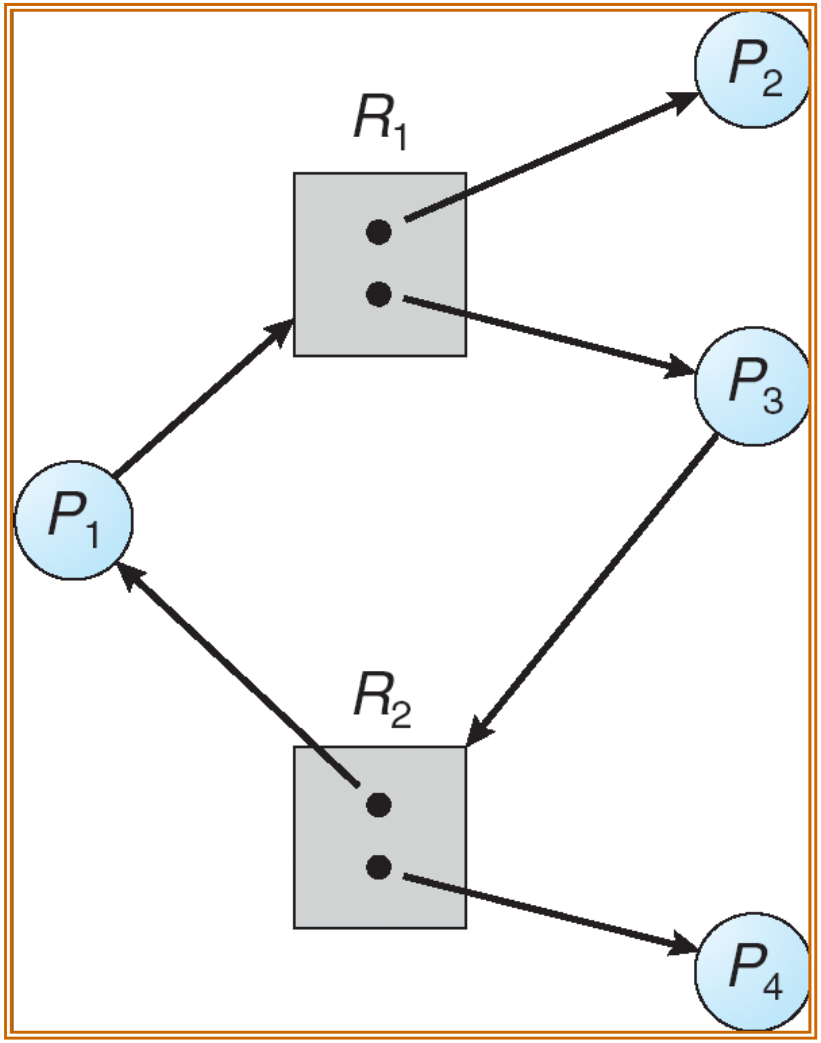


# Resource Allocation Graph With A Deadlock



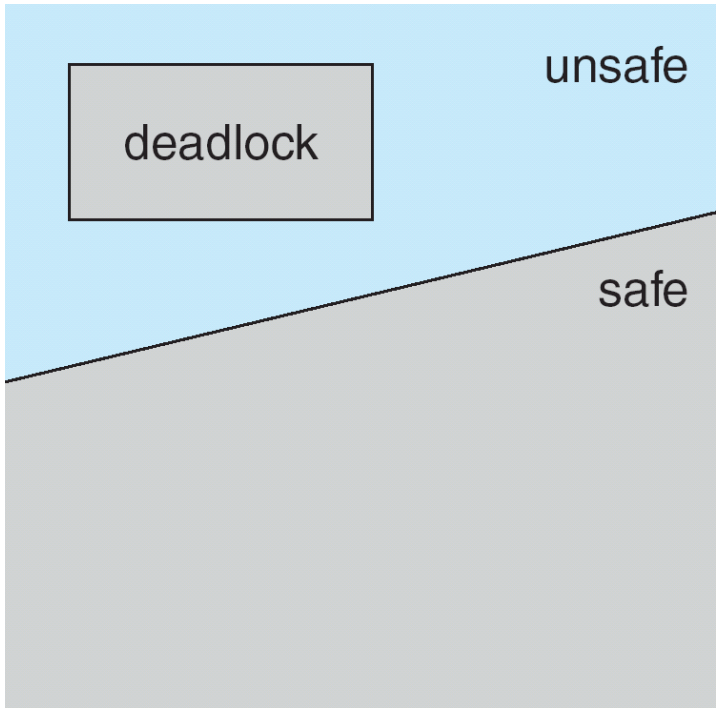
# Resource Allocation Graph Without A Deadlock

Even though there is a cycle, but some processes are not in the state of hold and wait ($P_2$ and $P_4$)




# Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a ***safe state***.
- System is in safe state if there exists a safe sequence of all processes to finish.
- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ deadlock is possible
- **Deadlock Avoidance** ⇒ ensure that a system will never enter an unsafe state.




# Safe and Unsafe States Example

Example 1: Safe (why?)

System has 12 total resources (currently available = 12-5-2-2=3)

| | Max Needed to finish | Currently Held |
|---|---|---|
| P0: | 10 | 5 |
| P1: | 4 | 2 |
| P2: | 9 | 2 |

Example 2: Unsafe unless a process releases what it is currently holding (why?)

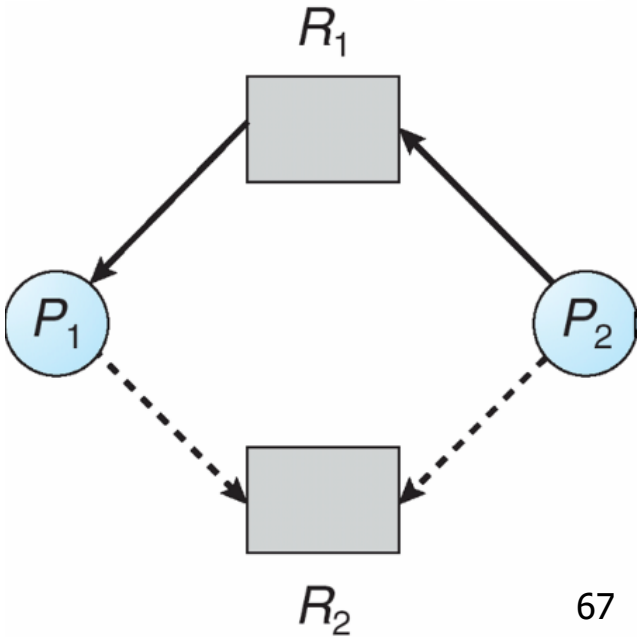
System has 12 total resources (currently available = 12-5-2-3=2)

| | Max Needed to finish | Currently Held |
|---|---|---|
| P0: | 10 | 5 |
| P1: | 4 | 2 |
| P2: | 9 | 3 (this is the only difference) |

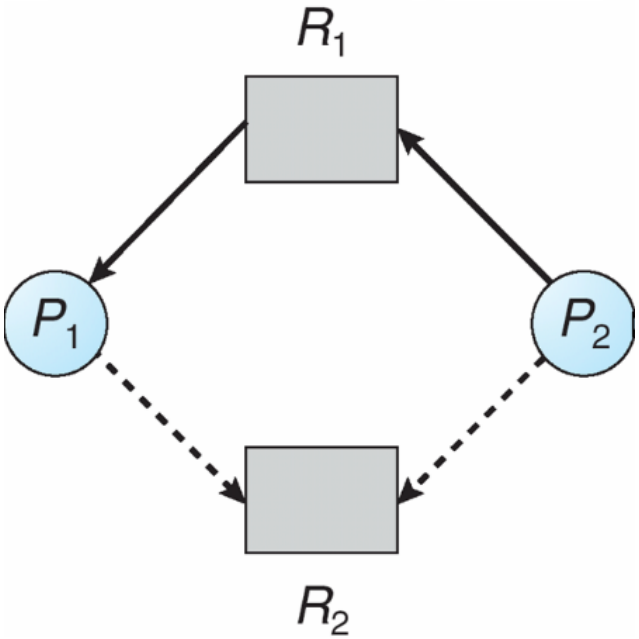
# RAG with Claims Edges (1/2)

- Claim edge $P_i \rightarrow R_j$ indicated that process $P_i$ may request resource $R_j$ in the future; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource and waits for it.
- Request edge converted to an assignment edge when the resource is allocated to the process.
- Resources must be claimed *a priori* in the system.

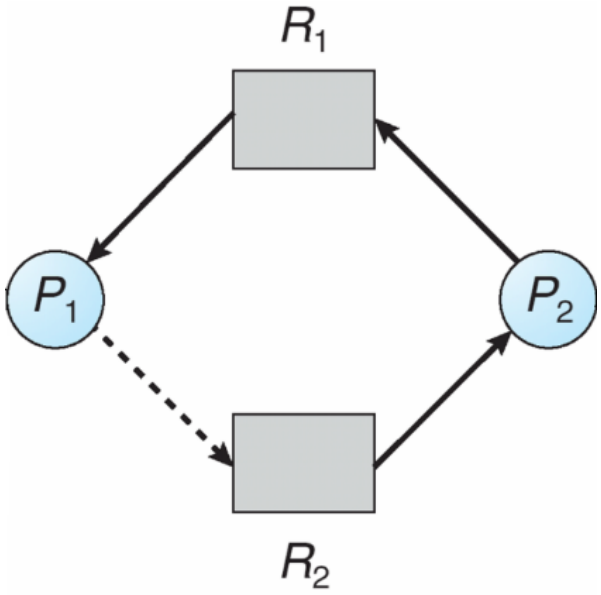
$P_2$ may request $R_2$
($R_2$ still free)



# RAG with Claims Edges (2/2)




$P_2$ may request $R_2$
($R_2$ still free)




Unsafe state

**Algorithm**: Request can be granted only if converting request into assignment does not lead to an unsafe state.


# Banker's Algorithm

- When a resource contains more than one instance the resource-allocation graph method does not work as there might be a cycle but no deadlock.
- The **Banker's Algorithm** (**BA**) gets its name because it is a method that bankers could use to assure that when they lend out cash, they will still be able to satisfy all their clients.
- When a thread starts up, it must state in advance the **maximum allocation of resources** it may request, **up to** the amount available on the system.
- When a request is made, the scheduler determines whether granting the request would leave the system in a **safe state**. If not, then the **thread must wait** until the request can be granted safely.

# BA Data Structures

Let ***p*** = number of **processes**, and ***r*** = number of **resources** types.

- ***Available:*** Vector of length ***r***. If ***available*[*j*]** = ***k***, there are ***k*** instances of resource type $R_j$ available.
- ***Max:*** ***p*** *x* ***r*** matrix. If ***Max***[***i,j***] = ***k***, then process $P_i$ may request at most ***k*** instances of resource type $R_j$.
- ***Allocation:*** ***p*** x ***r*** matrix. If ***Allocation***[***i,j***] = ***k*** then $P_i$ is currently allocated ***k*** instances of $R_j$.
- ***Need:*** ***p*** x ***r*** matrix. If ***Need***[***i,j***] = ***k***, then $P_i$ may need ***k*** more instances of $R_j$ to complete its task.

$$Need[i,j] = Max[i,j] - Allocation[i,j].$$


# BA Data Structures Example

- 5 processes $P_0$ through $P_4$ → (***p*** = 5)
- 3 resource types ***A*** (10 instances), ***B*** (5 instances), and ***C*** (7 instances) → (***r*** = 3)
- **State** of the data structures at time $T_0$:

| | *Allocation* | *Max* | *Available* | *Need* |
|---|---|---|---|---|
| | *A B C* | *A B C* | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 7 5 3 | 3 3 2 | 7 4 3 |
| $P_1$ | 2 0 0 | 3 2 2 | | 1 2 2 |
| $P_2$ | 3 0 2 | 9 0 2 | | 6 0 0 |
| $P_3$ | 2 1 1 | 2 2 2 | | 0 1 1 |
| $P_4$ | 0 0 2 | 4 3 3 | | 4 3 1 |


# BA Safety Algorithm

- The following safety algorithm is for determining whether a particular **state** is safe.

1. Let ***Work*** and ***Finish*** be vectors of length ***r*** and ***p*** respectively.
   - *Work* is a working copy of the *Available* resources, which will be modified during the analysis.
   - *Finish* is a vector of Booleans indicating whether a particular process can finish. ( or has finished so far in the analysis. )
   - Initialize *Work* to *Available*, and *Finish* to *False* for all processes.
2. Find an *i* such that both *Finish*[*i*] is *False* and *Need*[*i,j*] < *Work*[*j*] (for *j = 1 to r*). This indicates process *i* has not finished yet but could with the given available working set. If no such *i* exists, go to step 4.
3. Set *Work*[*j*] = *Work*[*j*] + *Allocation*[*i*, *j*] (for *j = 1 to r*) and set *Finish*[i] to *True*. This corresponds to process *i* finishing up and releasing its resources back into the work pool. Then loop back to step 2.
4. If *Finish*[*i*] is *True* for all *i*, then the state is a **safe** state, because a safe sequence has been found, otherwise the state is **unsafe**.

# BA Safety Algorithm Example

- 5 processes $P_0$ through $P_4$;
- 3 resource types *A* (10 instances), *B* (5 instances), and *C* (7 instances).
- **State** of the data structures at time $T_0$:

| | *Allocation* | *Max* | *Available* | *Need* |
|---|---|---|---|---|
| | *A B C* | *A B C* | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 7 5 3 | 3 3 2 | 7 4 3 |
| $P_1$ | 2 0 0 | 3 2 2 | | 1 2 2 |
| $P_2$ | 3 0 2 | 9 0 2 | | 6 0 0 |
| $P_3$ | 2 1 1 | 2 2 2 | | 0 1 1 |
| $P_4$ | 0 0 2 | 4 3 3 | | 4 3 1 |

- By applying the safety algorithm, the system is in a safe state since the sequence $P_1$, $P_3$, $P_4$, $P_2$, $P_0$ satisfies safety criteria


# BA Resource-Request Algorithm (1/2)

- This algorithm determines if a new request is safe and grants it only if it is safe to do so.

- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request.


# BA Resource-Request Algorithm (2/2)

1. Let ***Request****: **p** x **r*** matrix where If ***Request***[***i, j***] = ***k***, indicates that process $P_i$ requests ***k*** instances of resource type $R_j$.
   1. If ***k*** > ***Need***[***i, j***] (for any *j = 1 to r*), raise an error condition and stop.
   2. If ***k*** > ***Available***[***j***] (for any *j = 1 to r*), then that process must wait for resources to become available and stop.
2. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe (using the Safety Algorithm). If so, grant the request, and if not, then the process must wait until its request can be granted safely. The procedure for granting a request ( or pretending to for testing purposes ) is:
   - Available = Available - Request
   - Allocation = Allocation + Request
   - Need = Need - Request

# BA Resource-Request Example

- For the previous example assume a request from $P_1$ of (1, 0 , 2)
  - Recall $P_1$ Allocation, Available, and $P_1$ Need were 2 0 0, 3 3 2, and 1 2 2 respectively.
- As the Request ≤ Available, that is, (1,0,2) ≤ (3,3,2), then grant the request and update the state as shown below:

| | *Allocation* | *Max* | *Available* | *Need* |
|---|---|---|---|---|
| | *A B C* | *A B C* | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 7 5 3 | 2 3 0 | 7 4 3 |
| $P_1$ | 3 0 2 | 3 2 2 | | 0 2 0 |
| $P_2$ | 3 0 2 | 9 0 2 | | 6 0 0 |
| $P_3$ | 2 1 1 | 2 2 2 | | 0 1 1 |
| $P_4$ | 0 0 2 | 4 3 3 | | 4 3 1 |

- By applying the safety algorithm, the system is in a safe state since the sequence $P_1$, $P_3$, $P_4$, $P_0$, $P_2$ satisfies safety criteria.


# Readings and Resources List

- From Arpaci-Dusseau textbook:
  - Chapter 27: Interlude: Thread API