

---

# Project 1: Thread Scheduling

# Introduction to Pintos

---

- ❑ Simple OS for the 80x86 architecture
- ❑ Capable of running on real hardware
- ❑ We use bochs and/or qemu to run Pintos
- ❑ Provided implementation supports kernel threads, user programs and file system
- ❑ In the projects, strengthen support for these and implement support for virtual memory

# Version (Revision) Control System

- Start by choosing a code keeper for your group
- Keeper creates repository on CIS GForge, code.google.com, github.com, or sourceforge.net
- Base versions are checked out to create a working copy, and changes to working copy are committed using subversion, git, or some other version control system software.

The screenshot shows a Mozilla Firefox browser window displaying the MyGForge website. The address bar shows the URL <http://projects.cis.ksu.edu/gf/project/pintos/>. The page title is "MyGForge > Projects > Pintos Reference Implementation > Home". The browser's Most Visited list includes "Getting Started" and "Latest Headlines".

The MyGForge interface features a blue header with the "FORGE" logo and navigation links: Home, My Stuff, Search, Projects (selected), and Snippets. A user is logged in as "Neilsen" with a "Log Out" link. The breadcrumb trail is "Home > Projects > Pintos Reference Implementation > Home".

On the left, a sidebar for "Pintos Reference Implementation" includes a "Summary" link and a list of other links: Admin, Reporting, Search, Forums, and Tracker.

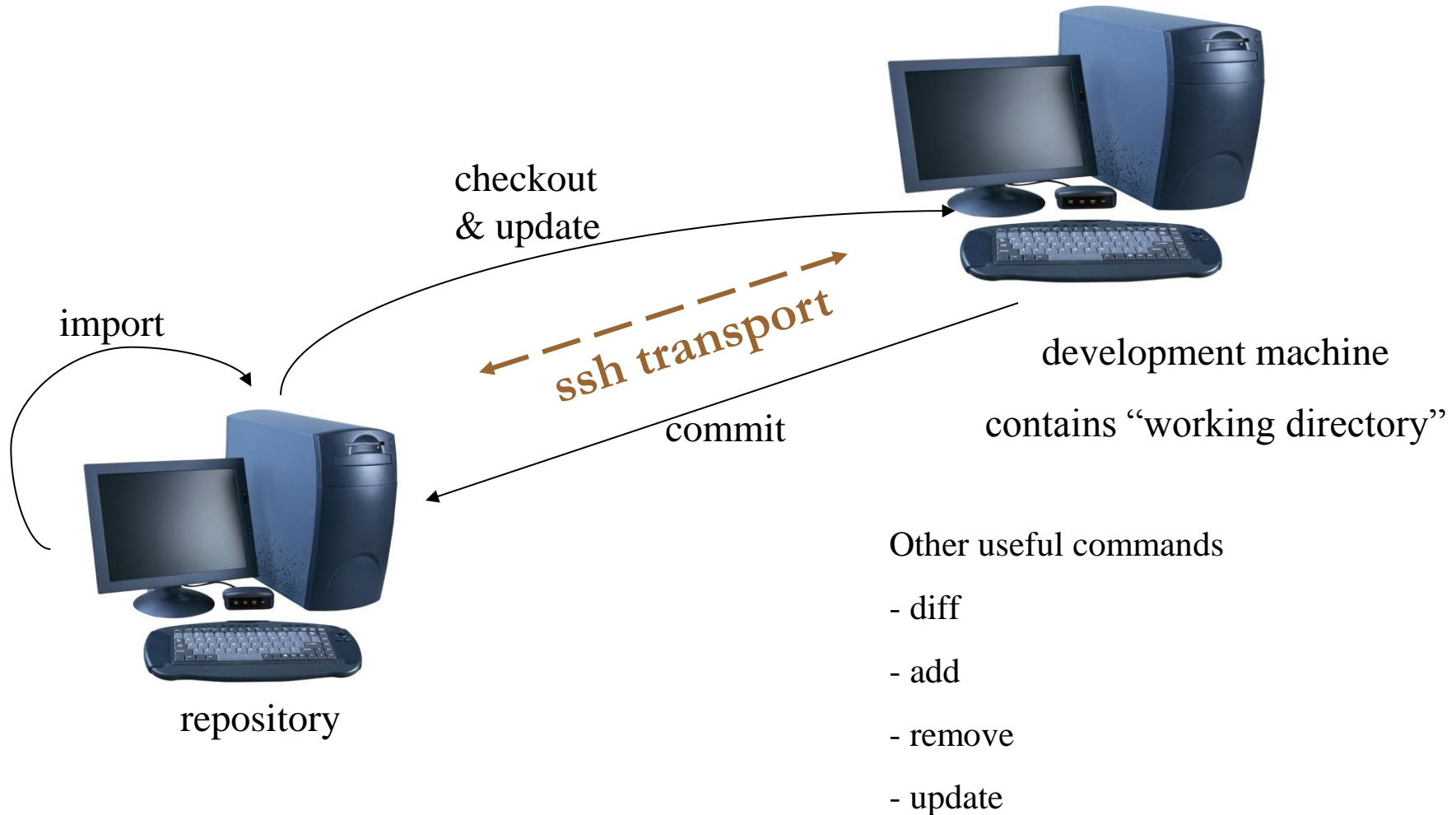
The main content area has a "Recent News" section stating "No news items found". Below this is a table of recent activities:

Time	Activity Type	By
2010-Sep-09		
11:45:23	<a href="#">Commit: Saving Stanford Code Expanded</a>	<a href="#">Mitchell Neilsen</a>
11:41:08	<a href="#">Commit: Saving Stanford Reference</a>	<a href="#">Mitchell Neilsen</a>

On the right, an "Activity" graph shows a line plot of activity over time, with the x-axis representing dates from 08-26 to 09-09 and the y-axis representing a numerical value from 0.0 to 1.0.

# Using Version Control System

---



# CVS Jargon

---

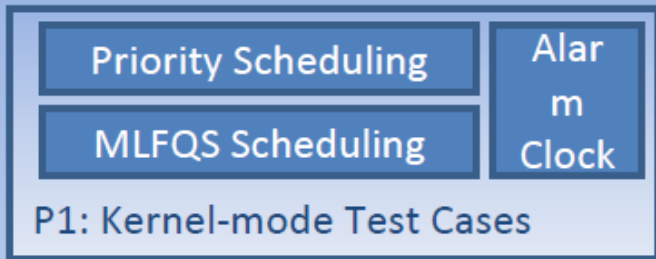
- “Do an update”
  - “Pull the latest”
  - “Commit your stuff”
  - “Push your changes”
  - “Diff against the HEAD”
  - “Diff against BASE”
  - “Outstanding diffs?”
- Bring your working directory in sync with the CVS repository to pick up and integrate changes other team members may have made.
- Upload your change to the CVS repository, allowing others to see them. May create a new revision if there were changes.
- Compare your working version to the version last checked in by any team member.
- Compare your working version to the version you last checked out. Any changes you’ve made are “outstanding” – group members can’t see them yet.

# Project 1 Overview

---

- ▣ Extend the functionality of a minimally functional thread system
- ▣ Implement
  - Alarm Clock
  - Priority Scheduling
    - Including priority inheritance (priority donation)
  - Advanced MLFQ Scheduler [Extra Credit]

# Project 1: Components



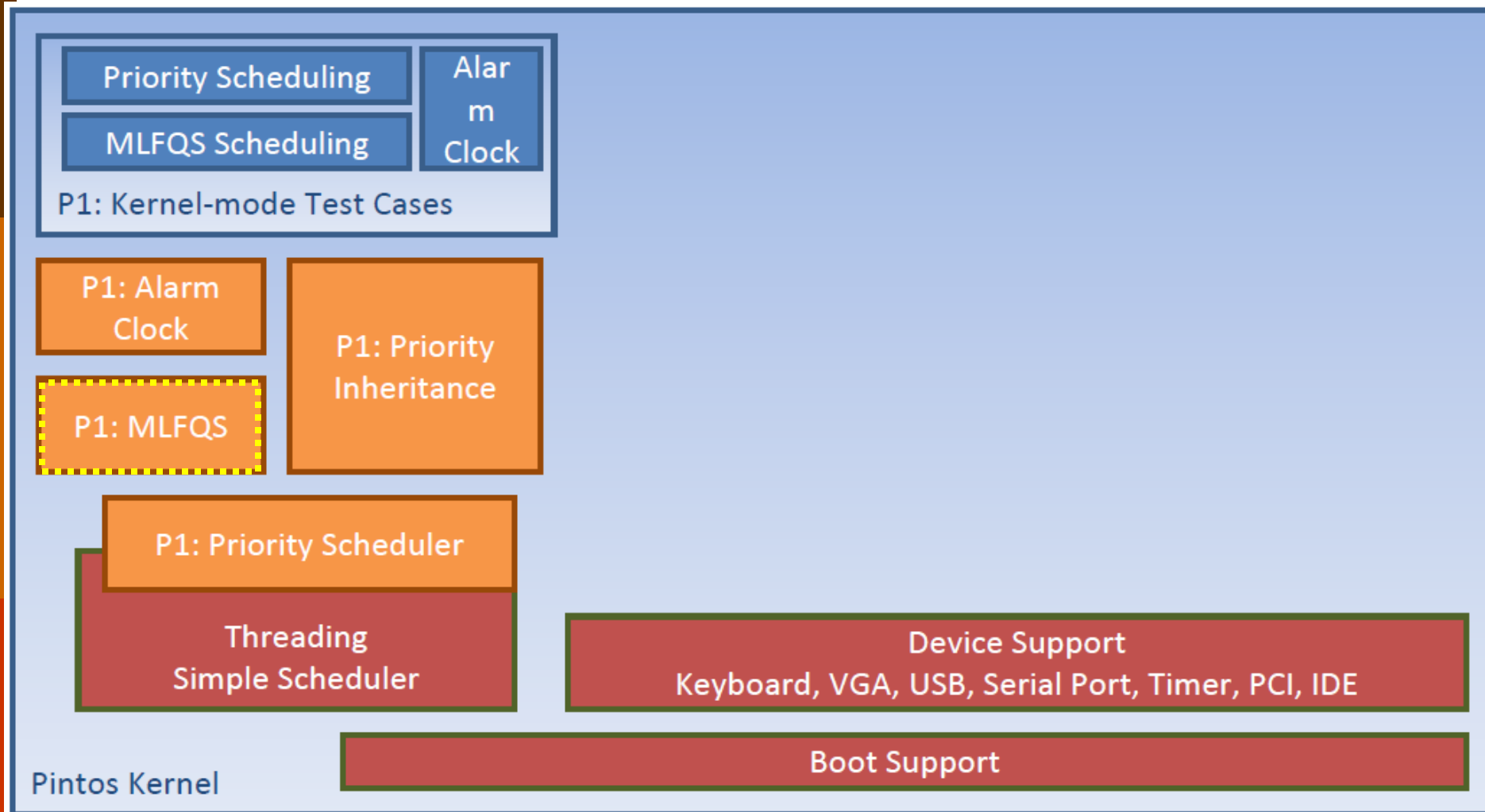
Threading  
Simple Scheduler

Device Support  
Keyboard, VGA, USB, Serial Port, Timer, PCI, IDE

Pintos Kernel

Boot Support

# Project 1: Components





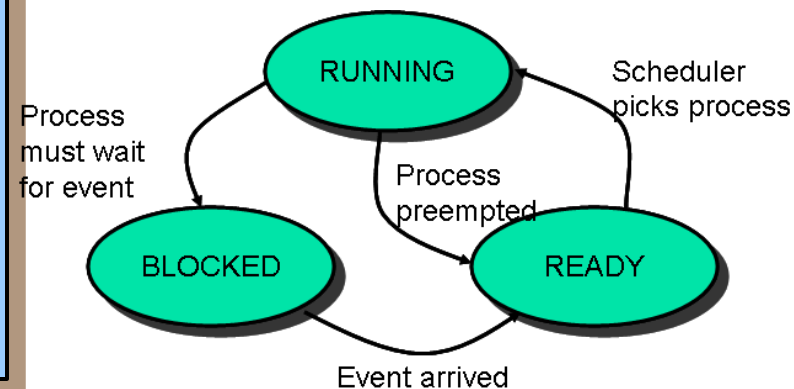
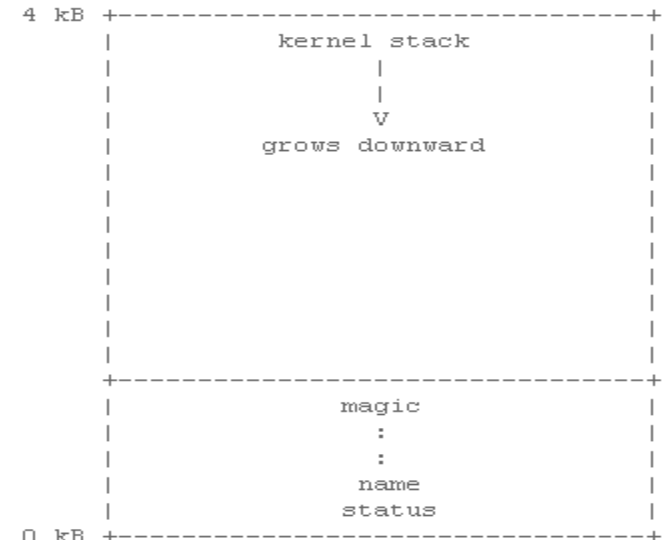
# Pintos Thread System

## src/threads/thread.h

```
struct thread
{
    tid_t tid;          /* Thread identifier. */
    enum thread_status status; /* Thread state. */
    char name[16]; /* Name (for debugging purposes). */
    uint8_t *stack; /* Saved stack pointer. */
    int priority; /* Priority. */
    struct list_elem allelem; /* List element for all-threads list. */
    /* Shared between thread.c and synch.c. */
    struct list_elem elem; /* List element. */
};
```

You add more fields here as you need them.

```
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir; /* Page directory. */
#endif
    /* Owned by thread.c. */
    unsigned magic; /* Detects stack overflow. */
};
```



# Pintos Thread System

## src/threads/thread.c

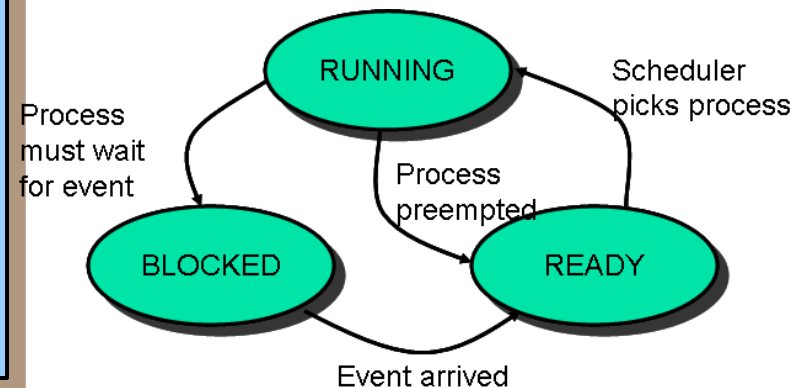
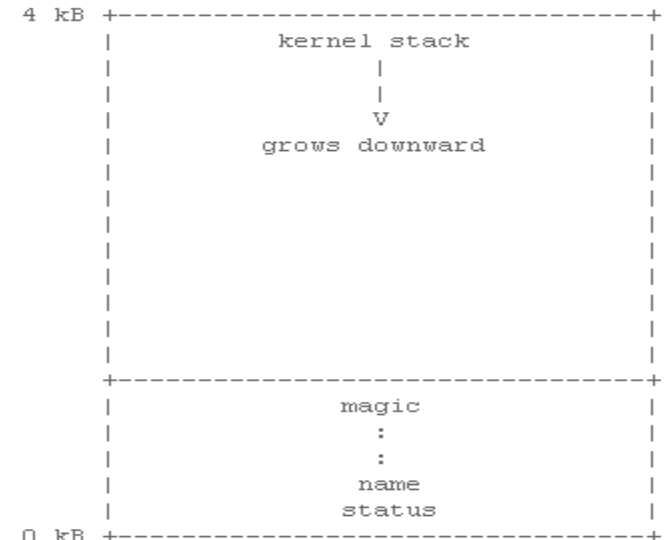
```
..
/* Random value for struct thread's `magic' member.
   Used to detect stack overflow. See the big comment at the top
   of thread.h for details. */
#define THREAD_MAGIC 0xcd6abf4b

/* List of processes in THREAD_READY state, that is, processes
   that are ready to run but not actually running. */
static struct list ready_list;

/* List of all processes. Processes are added to this list
   when they are first scheduled and removed when they exit. */
static struct list all_list;

/* Idle thread. */
static struct thread *idle_thread;
```

See src/lib/kernel/list.c for list handling functions



# Pintos Thread System (contd...)

---

- ▣ Read `threads/thread.c`, `threads/switch.S`, and `threads/synch.c` to understand
  - How the switching between threads occur
  - How the provided scheduler works
  - How the various synchronizations primitives work

# Alarm Clock

---

- ❑ Reimplement `timer_sleep( )` in `devices/timer.c` without busy waiting

```
/* Suspends execution for approximately TICKS timer ticks. */
```

```
void timer_sleep (int64_t ticks){  
    int64_t start = timer_ticks ();  
    ASSERT (intr_get_level () == INTR_ON);  
    while (timer_elapsed (start) < ticks)  
        thread_yield ();  
}
```

- ❑ Implementation details

- Remove thread from ready list and put it back after sufficient ticks have elapsed
- Use semaphore to block thread on semaphore associated with thread getting blocked; e.g., initialize semaphore with count?

# Semaphores [Dijkstra]

---

- **A *Semaphore* is initialized with an integer  $N$**
- **Provides two functions:**
  - `sem_wait (S)` (originally called  $P$ , called ***sema\_down*** in **Pintos**)
  - `sem_signal (S)` (originally called  $V$ , called ***sema\_up*** in **Pintos**)
- **Guarantees `sem_wait` will return only  $N$  more times than `sem_signal` called**
  - Example: If  $N == 1$ , then semaphore is a mutex with `sem_wait` as lock and `sem_signal` as unlock
- **Semaphores allow elegant solutions to some problems**

# Semaphore

---

A semaphore is a structure consisting of 2 parts:

```
struct semaphore {  
    int count; // number of resources available  
    queue Q; // queue of process/thread ids of blocked  
}
```

Shorthand notation:

semaphore  $S = 1 \rightarrow S.count = 1, S.Q = \{ \}$

# Operations on Semaphores

---

There are two basic semaphore operations:

`sem_wait(S):`

- if ( $S.count > 0$ ) then  $S.count = S.count - 1$ ;
- else block calling process in  $S.Q$ ;

`sem_signal(S):`

- if ( $S.Q$  is non-empty) then wakeup a process in  $S.Q$ ;
- else  $S.count = S.count + 1$ ;

# Semaphore Example: Mutual Exclusion

---

Semaphore  $S = 1$ ;

Thread A:

```
sem_wait(S);  
    (do work in critical section CS);  
sem_signal(S);
```

Thread B:

```
sem_wait(S);  
    (do work in CS);  
sem_signal(S);
```



# Semaphore Example: Order Execution

---

Semaphore S = 0;

Thread A → Thread B:

Thread A:

(do work);

sem\_signal(S);

Thread B:

sem\_wait(S);

(do work);

# Pintos Semaphores

---

- ❑ `struct semaphore s;`
- ❑ `sema_init(&s, 1);`
- ❑ `sema_down(&s);`
- ❑ `sema_up(&s);`

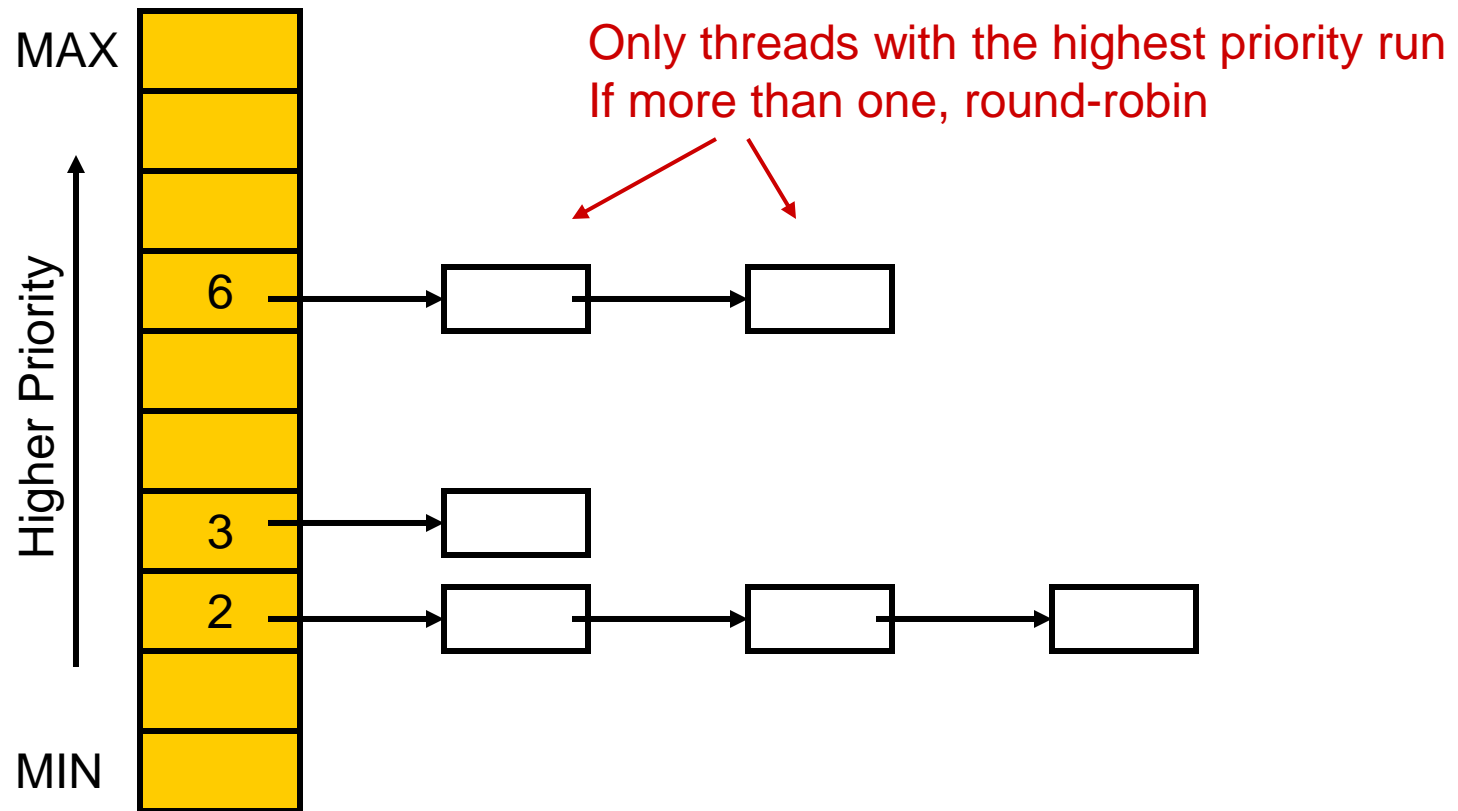
# Priority Scheduler

---

- ❑ Ready thread with highest priority gets the processor
- ❑ When a thread is added to the ready list that has a higher priority than the currently running thread, immediately yield the processor to the new thread
- ❑ When threads are waiting for a lock, semaphore or a condition variable, the highest priority waiting thread should be woken up first
- ❑ Implementation details
  - compare priority of the thread being added to the ready list with that of the running thread
  - select next thread to run based on priorities
  - compare priorities of waiting threads when releasing locks, semaphores, condition variables

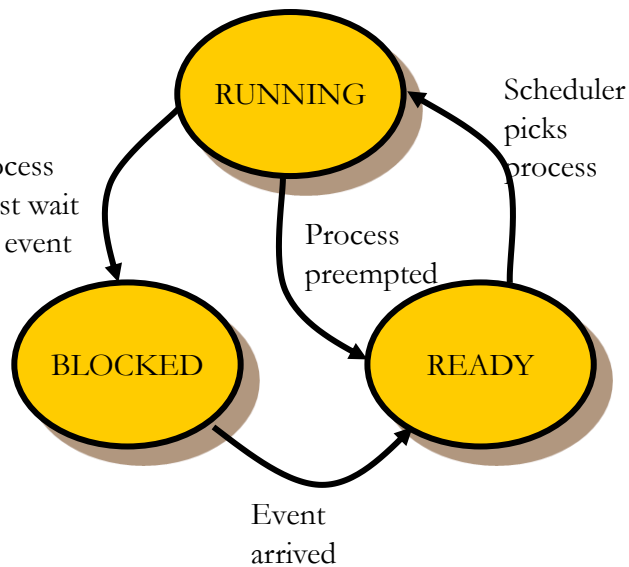
# Priority Based Scheduling

---



# Using `thread_yield()` to implement preemption

---



- ❑ Current thread ("RUNNING") is moved to READY state, added to READY list.
- ❑ Then scheduler is invoked. Picks a new READY thread from READY list.
- ❑ Case a): there's only 1 READY thread. Thread is rescheduled right away
- ❑ Case b): there are other READY thread(s)
  - b.1) another thread has higher priority – it is scheduled
  - b.2) another thread has same priority – it is scheduled provided the previously running thread was inserted in tail of ready list.
- ❑ "thread\_yield()" is a call you can use whenever you identify a need to preempt current thread.
- ❑ **Exception:** inside an interrupt handler, use "intr\_yield\_on\_return()" instead

# Priority Inversion

---

- ❑ Strict priority scheduling can lead to a phenomenon called “priority inversion”
- ❑ Supplemental reading:
  - What really happened on the Mars Pathfinder?
- ❑ Consider the following example where  $\text{prio}(H) > \text{prio}(M) > \text{prio}(L)$ 
  - H needs a lock currently held by L, so H blocks
  - M that was already on the ready list gets the processor before L
  - H indirectly waits for M
  - (on Path Finder, a watchdog timer noticed that H failed to run for some time, and continuously reset the system)

# Priority Donation

---

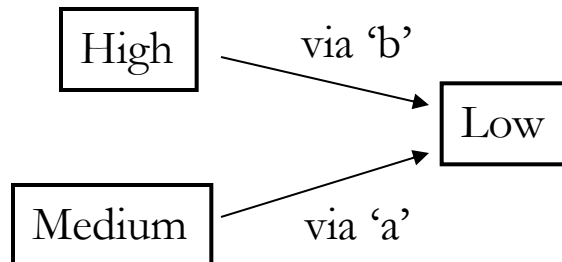
- ❑ When a high priority thread H waits on a lock held by a lower priority thread L, donate H's priority to L and recall the donation once L releases the lock
- ❑ Implement priority donation for locks
- ❑ Handle the cases of multiple donations and nested donations

# Multiple Priority Donations: Example

---

## Low Priority thread

```
lock_acquire (&a);  
lock_acquire (&b);  
  
thread_create ("a", PRI_DEFAULT + 1, a_thread_func, &a);  
msg ("Main thread should have priority %d. Actual priority:  
%d.", PRI_DEFAULT + 1, thread_get_priority ());  
  
thread_create ("b", PRI_DEFAULT + 2, b_thread_func, &b);  
msg ("Main thread should have priority %d. Actual priority:  
%d.", PRI_DEFAULT + 2, thread_get_priority ());
```



## Medium Priority thread

```
static void a_thread_func (void *lock_)  
{  
    struct lock *lock = lock_;  
    lock_acquire (lock);  
    msg ("Thread a acquired lock a.");  
    lock_release (lock);  
    msg ("Thread a finished.");  
}
```

## High Priority thread

```
static void b_thread_func (void *lock_)  
{  
    struct lock *lock = lock_;  
    lock_acquire (lock);  
    msg ("Thread b acquired lock b.");  
    lock_release (lock);  
    msg ("Thread b finished.");  
}
```



# Nested Priority Donations: Example

## Low Priority thread

```
lock_acquire (&a);
locks.a = &a;
locks.b = &b;

thread_create ("medium", PRI_DEFAULT + 1, m_thread_func, &locks);
msg ("Low thread should have priority %d. Actual priority: %d.",
PRI_DEFAULT + 1, thread_get_priority ());

thread_create ("high", PRI_DEFAULT + 2, h_thread_func, &b);
msg ("Low thread should have priority %d. Actual priority: %d.",
PRI_DEFAULT + 2, thread_get_priority ());
```

## Medium Priority thread

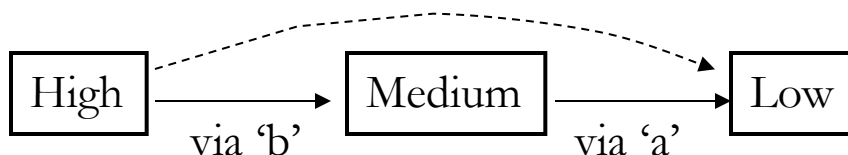
```
static void m_thread_func (void *locks_)
{
    struct locks *locks = locks_;
    lock_acquire (locks->b);
    lock_acquire (locks->a);

    msg ("Medium thread should have priority %d.
Actual priority: %d.", PRI_DEFAULT + 2,
        thread_get_priority ());
    ...}
}
```

## High Priority thread

```
static void h_thread_func (void *lock_)
{
    struct lock *lock = lock_;

    lock_acquire (lock);
    ...}
}
```



# Advanced Scheduler [Extra Credit]

---

- ❑ Implement Multi Level Feedback Queue Scheduler
- ❑ Priority donation not needed in the advanced scheduler – two implementations are not required to coexist
  - Only one is active at a time
- ❑ Advanced Scheduler must be chosen only if `'-mlfqs'` kernel option is specified
- ❑ Read section on 4.4 BSD Scheduler in the Pintos manual for detailed information
- ❑ Some of the parameters are real numbers and calculations involving them have to be simulated using integers.
  - Write a fixed-point layer (header file)

# Typesafe Fixed-Point Layer

---

```
typedef struct
{
    double re;
    double im;
} complex_t;
```

```
static inline complex_t
complex_add(complex_t x, complex_t y)
{
    return (complex_t){ x.re + y.re, x.im + y.im };
}
```

```
static inline double
complex_real(complex_t x)
{
    return x.re;
}
```

```
static inline double
complex_imaginary(complex_t x)
{
    return x.im;
}
```

```
static inline double
complex_abs(complex_t x)
{
    return sqrt(x.re * x.re + x.im * x.im);
}
```

# Suggested Order

---

- ❑ Alarm Clock
  - easier to implement compared to the other parts
  - other parts not dependent on this
- ❑ Priority Scheduler
  - needed for implementing Priority Donation and Advanced Scheduler
- ❑ Priority Donation | Advanced Scheduler
  - these two parts are independent of each other
  - can be implemented in any order but only after Priority Scheduler is ready

# Debugging your code

---

▣ printf, ASSERT, backtraces, gdb

▣ Running pintos under gdb

- Invoke pintos with the gdb option

`pintos --gdb -- run testname`

- On another terminal invoke gdb

`pintos-gdb kernel.o`

- Issue the command

`debugpintos`

- All the usual gdb commands can be used: step, next, print, continue, break, clear etc
- Use the pintos debugging macros described in manual

# Tips

---

- ❑ Read the relevant parts of the Pintos manual
- ❑ Read the comments in the source files to understand what a function does and what its prerequisites are
- ❑ Be careful with synchronization primitives
  - disable interrupts only when absolutely needed
  - use locks, semaphores and condition variables instead
- ❑ Beware of the consequences of the changes you introduce
  - might affect the code that gets executed before the boot time messages are displayed, causing the system to reboot or not boot at all

# Tips (contd...)

---

- ❑ Include ASSERTs to make sure that your code works the way you want it to
- ❑ Integrate your team's code often to avoid surprises
- ❑ Use gdb to debug
- ❑ Make changes to the test files, if needed
- ❑ Test using qemu simulator and the -j option with bochs (introduces variability whereas default options run in reproducibility mode)

# Grading

---

- ▣ Tests – 50%
  - All group members get the same grade
- ▣ Design – 50%
  - Data structures, algorithms, synchronization, rationale and coding standards
  - Each team member should understand all data structures, algorithms, and rationale for implementation

Good Luck!