Condition Variables

Mythili Vutukuru
CSE, IIT Bombay

Recap: Socket communication

- Client and server applications on the Internet communicate with each other using sockets, e.g., web browser and web server
  - Server opens socket on a well known address and starts listening
  - Client opens a socket and connects to server's socket
  - Client and server exchange requests and responses

Client

```
sockfd = socket(..)
connect(sockfd, server_sockaddr, ..)
n = send(sockfd, req_buf, req_len, ..)
n = recv(sockfd, resp_buf, resp_len, ..)
```

Server

```
sockfd = socket(..)
bind(sockfd, server_address)
listen(sockfd, ..)
newsockfd = accept(sockfd, ..)
n = recv(newsockfd, req_buf, req_len, ..)
n = send(newsockfd, resp_buf, resp_len, ..)
```

Multi-threaded applications

- Real-life applications do multiple things concurrently
  - Server has to listen for and accept new connections from new clients
  - Server has to handle requests coming in from existing clients
  - These various operations may block (e.g., read blocks till data arrives)
- Real-life applications have multi-threaded designs, e.g., master-worker
  - Main master thread of server accepts new connections, places new connections or requests in a shared queue
  - Worker threads pick requests from the queue one by one, and service them
  - Mutual exclusion using locks when adding/removing requests from queue
- How does worker thread know when request has arrived in queue?
  - All worker threads constantly check the queue all the time? (inefficient polling)

What we need: wait and signal mechanism

- Locks allow one type of synchronization between threads – mutual exclusion when accessing critical sections
- Another common requirement in multi-threaded applications – waiting for events and signaling when event occurs
  - E.g., Worker thread wants to run only after master thread has placed a new request in the shared queue
  - E.g., Thread T2 wants to run only after T1 has finished some task (T1→T2)
- Naive solution: T2 keeps checking periodically if T1 is done
  - Wastes CPU cycles, inefficient
  - Need a new synchronization primitive to wait for an event

Condition variables

- Pthread library provides special variables called condition variables (CV)
  - A thread calls wait function on a CV, it is blocked and gets added to a list of threads waiting on that CV
  - Another thread calls signal on a CV, one of the waiting threads gets ready to run again, will be scheduled in the future (no immediate context switch)
- Example: we want T2 to run only after T1 does its work (T1→T2)
  - T1 does its work and calls signal
  - T2 checks if work is done, and calls wait if work is not done

T2

```
if(!done) wait(cv)
(T2 blocks)

(T2 resumes)
```

T1

```
//do work
done = true
signal(cv)
```

T1

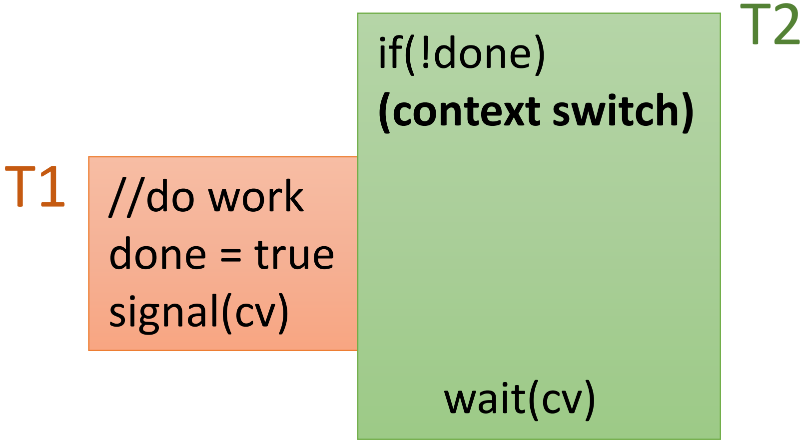
```
//do work
done = true
signal(cv)
```

T2

```
if(!done) wait(cv)
//proceed, no wait
```

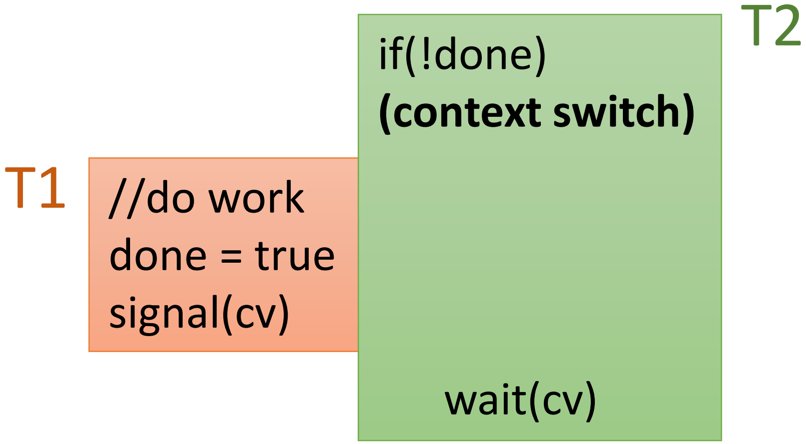
Atomicity in wait and signal (1)

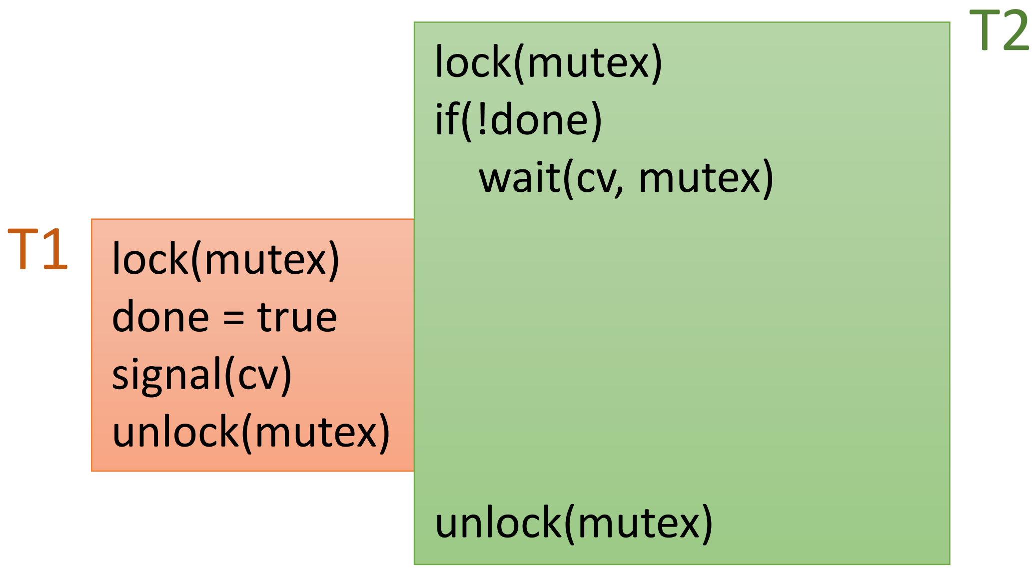
- Checking condition and waiting must be atomic, deadlock otherwise
  - Thread T2 checks condition is false, context switch just before blocking
  - Meanwhile T1 makes condition true, calls signal. But signal doesn’t wake up anyone (none sleeping yet)
  - T2 resumes, goes to sleep forever (no one will signal again)
- This is called missed wakeup problem: how to fix?



Atomicity in wait and signal (2)

- Solution: use a lock/mutex to protect atomicity of sleeping
  - T2 holds a lock, checks condition, calls wait
  - Lock released only after T2 is added to list of waiting processes (ensures atomicity of checking condition and sleeping)
  - T1 acquires **same** lock before calling signal, ensuring that signal cannot happen in between checking condition and waiting
  - Pthread CV implementation releases lock during wait, reacquires on wakeup





Example: parent waits for child

```
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)
        Pthread_cond_wait(&c, &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;
}
```

Figure 30.3: **Parent Waiting For Child: Use A Condition Variable**

Image credit: OSTEP

Guidelines for using condition variables

- Use the same lock for wait and signal (maybe for other variables too)
- Before calling wait, confirm that the condition is indeed false
  - T2 must check “done” variable before calling wait (what if T1 has already run?)
- Signal broadcast wakes up all threads while signal wakes up any one
- Good habit to check condition with “while” loop and not “if”
  - To avoid corner cases of thread being woken up even when condition not true (may be an issue with some implementations)

```
if(condition)
  wait(condvar)
//small chance that condition may be false when wait returns

while(condition)
  wait(condvar)
//condition guaranteed to be true since we check in while-loop
```

Example: Producer-consumer problem

- Producer and consumer threads, sharing data via a buffer of bounded size
  - Producers produce items, add into a shared buffer
  - Consumers consume item from shared buffer
- What kind of coordination is needed between threads?
  - Producer thread produces and places items into buffer, waits if the buffer is full → Consumer signals after making space in the buffer
  - Consumer thread consumes items from buffer, waits if the buffer is empty → Producer signals after producing items

Example: Multi-threaded server

- Master thread accepts requests and puts them in a queue
- Worker threads fetch requests from this queue and process them

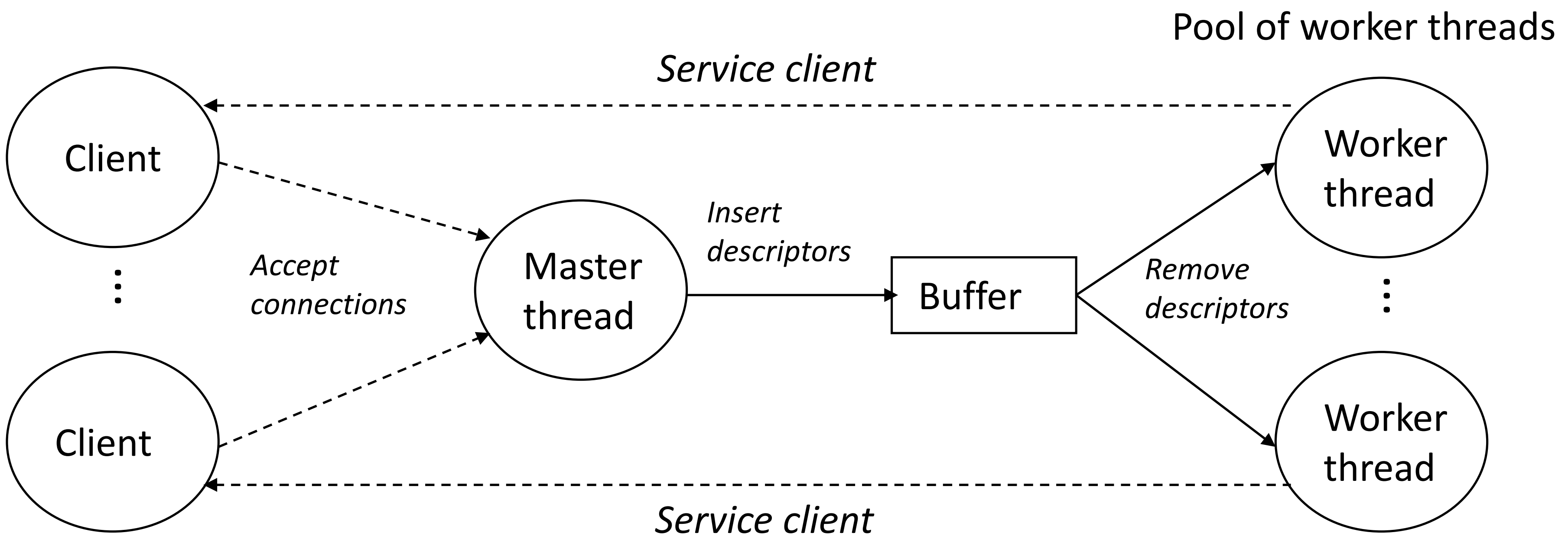


Image credit: CSAPP

Example: Producer-consumer problem

- Solution using condition variables
  - Mutex/lock used while modifying shared buffer
  - Two CVs: one for producers to wait, and one for consumers to wait

```
//Producer
lock(mutex)
if(no free space in buffer)
    wait(cv_producer, mutex)
produce item, add to buffer
signal(cv_consumer)
unlock(mutex)
```

```
//Consumer
lock(mutex)
if(no items in buffer)
    wait(cv_consumer, mutex)
consume item from buffer
signal(cv_producer)
unlock(mutex)
```

Producer/Consumer with 2 CVs

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == MAX)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

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

Image credit: OSTEP

Example: Batched processing

- Example scenario: two kinds of threads in an application
  - Request threads, each containing an application request
  - Batch processor thread processes N requests at a time in a batch
- What kind of synchronization do we need?
  - Batch processing thread must wait until N requests arrive, then start batch
  - Request thread must wait until batch starts, then get processed and finish
- Example: suppose Covid-19 vaccination vial has 10 doses. Nurse waits for 10 patients to arrive, then opens the vial and vaccinates all 10

Example: Batched processing

- Solution using two CVs: one for requests to wait, one for batch processor to wait
  - Other integer and Boolean variables, mutex/lock for atomicity

```
//Request thread
lock(mutex)
count++
if(count == N)
   signal(cv_batch_processor)
while(not batch_started)
   wait(cv_request, mutex)
unlock(mutex)
```

```
//Batch processor thread
lock(mutex)
while(count < N)
   wait(cv_batch_processor, mutex)
batch_started = true
signal_broadcast(cv_request)
unlock(mutex)
```

Example: Batched processing

- What is wrong with this solution?
  - Nth request thread calls wait before invoking signal to wake up batch processor
  - Batch processor never wakes up, all threads will sleep forever
  - Before you sleep, ensure that the signaling code can run in future

```
//Request thread
lock(mutex)
count++
while(not batch_started)
   wait(cv_request, mutex)
if(count == N)
   signal(cv_batch_processor)
unlock(mutex)
```

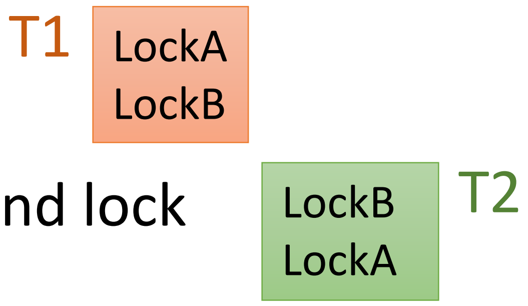
```
//Batch processor thread
lock(mutex)
while(count < N)
   wait(cv_batch_processor, mutex)
batch_started = true
signal_broadcast(cv_request)
unlock(mutex)
```

Synchronization patterns using CVs

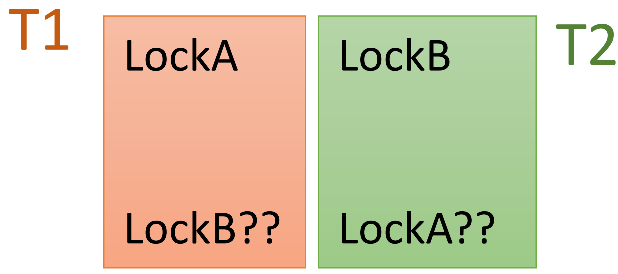
- Many examples in the practice problems
  - Scenario describing multiple threads/entities and how they should interact and coordinate with each other
  - Toy examples modelled after real world application design patterns
- How to write code with correct synchronization
  - Identify when each entity should wait and write the suitable waiting code
  - For each wait, figure out how the signaling will happen and write the code
  - Ensure that signaling path in the code is not blocked in any way, e.g., signal others first before calling wait and going to sleep
  - Update all extra variables (counts, flags) in the solution correctly
  - Run through your code in a few different scenarios and different order of execution of threads to convince yourself that it works correctly

Watch out for deadlocks

- Deadlock: threads are stuck in blocked state without making progress
- Example: thread sleeps by calling wait on CV, no other thread calls signal, so thread sleeps forever
- Example: circular wait when acquiring multiple locks
  - T1 acquires LockA and LockB, T2 acquires LockB and LockA
  - T1 acquires LockA, T2 acquires LockB, each is waiting for second lock
  - Deadlock if executions interleave in some ways



- Techniques to avoid deadlocks
  - Acquire locks in same order across all threads of process
  - When sleeping, ensure someone will wake you up!



Sleep and wakeup in xv6 (1)

- xv6 does not have userspace threads, only single threaded processes
- But multiple processes may be in kernel mode on different CPU
  - Uses locks to protect access to shared kernel data structures
- OS also needs a mechanism to let processes sleep (e.g., when process makes blocking disk read syscall) and wakeup when some events occur (e.g., disk has raised interrupt and data is ready)
- Process P1 in kernel mode calls sleep to give up CPU, gets blocked until event
- Another process P2 (in kernel mode) wakes up P1 when the event occurs

Sleep and wakeup in xv6 (2)

- A process P1 that wishes to block and give up CPU calls “sleep”
  - Example: process reads a block from disk, must block until disk read completes
  - Read syscall → sleep → sched() to give up CPU
- Another process P2 calls “wakeup” when event to unblock P1 occurs
  - P2 calls wakeup → marks P1 as runnable, no context switch immediately
  - Example: disk interrupt occurred when P2 is running, P2 runs interrupt handler, which will call wakeup
- P1 will be scheduled at a later time, will resume at sched(), return
- Spinlock protects atomicity of sleep: P1 calls sleep with some spinlock L held, P2 calls wakeup with same spinlock L held

Sleep and wakeup in xv6 (3)

- How does P2 know which process to wake up?
- When P1 sleeps, it sets a channel (void * chan) in its struct proc
  - Arguments to sleep: channel, spinlock to protect atomicity of sleep
- P2 calls wakeup on same channel
  - Arguments to wakeup: channel (lock must be held)
- Channel = any value known to both P1 and P2
  - Example: channel value for disk read can be address of disk block

Example: wait and exit

- If wait called in parent while children are running, parent calls sleep and gives up CPU (channel is parent struct proc pts, lock is ptable.lock)

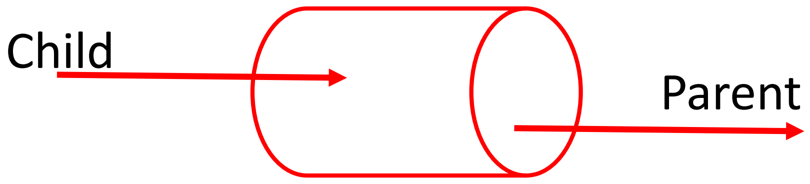
```
    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);
```

- In exit, child acquires ptable.lock, wakes up parent using its channel

```
// Parent might be sleeping in wait().
wakeup1(curproc->parent);
```

- Why is terminated process memory cleaned up by parent?
  - When a process calls exit, kernel stack, page table etc are in use, all this memory cannot be cleared until terminated process has been taken off the CPU

Example: pipes in xv6 (1)



- xv6 provides anonymous pipes for IPC between parent and child processes
- Example: Parent P and child C share anonymous pipe
- Child C writes into pipe, parent P reads from pipe
- One of P or C closes read end, other closes write end

```
//userspace code

int fd[2]
pipe(fd) //syscall to create pipe

int ret = fork()

if(ret == 0) {//child
   close(fd[0]) //close read end
   write(fd[1], message, ..)
}
else {//parent
   close(fd[1]) //close write end
   read(fd[0], message, ..)
}
```

Example: pipes in xv6 (2)

- Internal implementation inside kernel
  - Common shared buffer, protected by a spinlock
  - Write system call stores data in shared buffer
  - Read system call returns data from shared buffer
  - Variables nread and nwrite indicate number of bytes read/written in buffer

```
struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;      // number of bytes read
  uint nwrite;     // number of bytes written
  int readopen;    // read fd is still open
  int writeopen;   // write fd is still open
};
```

Example: pipes in xv6 (3)

- Implementation of pipe read and write system calls uses sleep/wakeup
- Pipe reader sleeps if pipe is empty, pipe writer wakes it up
- Pipe writer sleeps if pipe is full, pipe reader wakes it up
- Channel for sleep/wakeup = address of pipe structure variables

```
int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE){
      if(p->readopen == 0 || myproc()->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);
  release(&p->lock);
  return n;
}
```

pipe is full

writer's channel for sleep is address of nwrite variable

Example: pipes in xv6 (4)

```
int
piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  while(p->nread == p->nwrite && p->writeopen){
    if(myproc()->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock);
  }
  for(i = 0; i < n; i++){
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
  wakeup(&p->nwrite);
  release(&p->lock);
  return i;
}
```

pipe is empty

reader’s channel is address of nread variable

pipe lock protects atomicity of sleep

Sleep function

- Sleep and wakeup called by processes with same lock held (to protect atomicity of sleep)
- Acquire ptable lock (if not already taken), then release other spinlock
- Reacquire original lock on return

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){
    acquire(&ptable.lock);
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  // Tidy up.
  p->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){
    release(&ptable.lock);
    acquire(lk);
  }
}
```

Wakeup function

- Wakeup acquires ptable.lock to change process to runnable
- If lock protecting atomicity of sleep is ptable.lock itself, then directly call wakeup1
- Wakes up all processes sleeping on a channel in ptable (more like signal broadcast of condition variables)

```
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```