SYSC 5701

Operating System Methods for Real-Time Applications

Monitors

Winter 2014

Motivation for Monitors

- Conflicting goals in real-time systems:
  application-specific behaviour **vs.**
  generic kernel support for process model
- Kernel may introduce unnecessary overhead
- Access to shared resources often involves passing synchronization gates to ensure access is possible (semaphore overhead)
- Every Wait call includes: call to o/s service and return → even if the Wait does not result in becoming blocked!

Recall Stream-2-Pipe Example

- **free_space** & **packets_in_Q** sema4s
- only really necessary to “wait” under certain conditions (no space OR no packets)
- if conditions could be “known” then could decide to wait only when necessary
- only call sema4 services when necessary? ☺
- Could some application-specific process management improve efficiency ?!

  (reduce overhead)

  → **MONITOR** !

Monitors

- application-specific protected services
- protected layer between kernel and rest of application
  - encapsulates critical section
- access only via entry procedures
- **passive** collection of procedures and data
- **"mutex constraint":** must **design** such that **mutually exclusive** execution inside monitor

KEY!

Consistent state!

Monitors

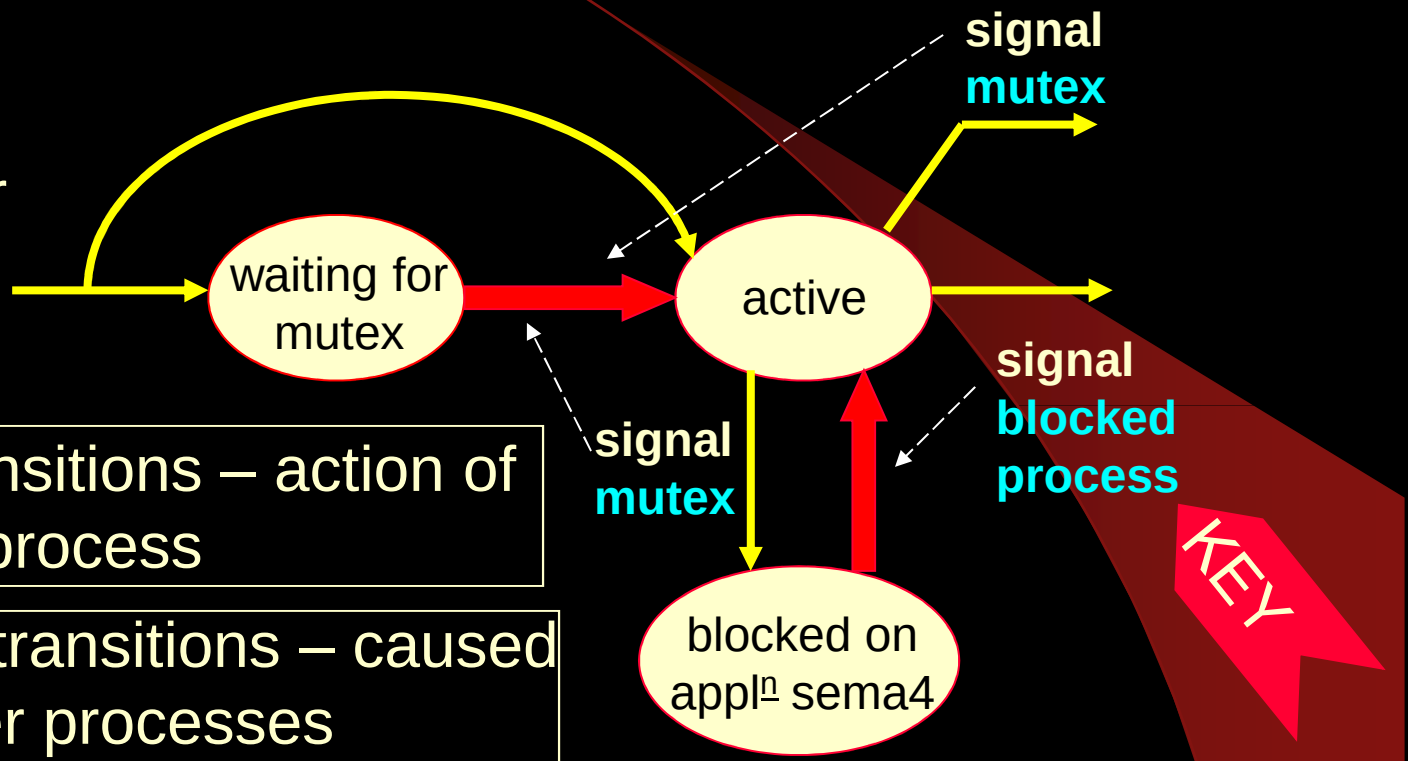
- allow reduction in use of kernel services
  - lower overhead ☺
  - require careful design ☹
  - mutex at all times! – crafty!
- to satisfy mutex constraint:

  → **additional kernel service**
  - if sema4s used, will need sema4 op:

wait_and_signal(wait_sema4, sig_sema4)

Generalized State Changes for Process in Monitor

Assume sema4:
mutex
Protects monitor



self transitions – action of active process

forced transitions – caused by other processes

effect the state of other processes!

Recall Semaphore-Based Synchronization Example

- Protected_**Add** to Q
- Protected_**Remove** from Q

included sema4s to check full/empty in **every** call

- consider monitor implementation structures:
  - **mutex** : sema4 = 1; // as before
  - will include 2 more sema4's (as before), **but** will only wait/signal **when necessary**
- frequent case: wait/signal **mutex only**
  - non-blocking → no other sema4s involved!

Q Full Objects

want_space : sema4 = 0;

// similar to before, BUT

// wait here only when no space in Q

waiting_4_space : integer = 0;

// NEW → local count of depth of want_space's

// blocked_Q i.e. value = # processes currently

// waiting at want_space

Q Empty Objects

want_work : sema4 = 0;

// wait here only when no work

waiting_4_work : integer = 0;

// value = # currently waiting at want_work

work_in_Q : integer = 0;

// value = # of packets currently in Q

Monitored_Add (P: packet_buffer)

```
{  mutex . Wait;                    // mutually exclusive access!
   if    work_in_Q  = =  Q_Size  // no space – must wait!
   {     waiting_4_space  =  waiting_4_space + 1;
         wait_and_signal ( want_space,  mutex );
         // new process enters monitor
         //       if add – ends up waiting here too, OR
         //       if remove – will free up a space, and then
         //       signal want_space!
   }  // process gets here eventually (owns mutex!)
                  // (continued on next slide …)
```

Slide 6?

```
Packet_Q . Add( P );          // add to Q
work_in_Q  =  work_in_Q  +  1;
// either signal a waiting process,
// or let in a new process
if  waiting_4_work  >  0
{    waiting_4_work  =  waiting_4_work  –  1;
     want_work . Signal ; // signal waiting process
     // leave without signaling mutex !!
} else {   mutex.Signal  } // let in a new process
}   // DONE!
```

KEY

Monitored_Remove **(var P: packet_buffer)**

```
{  mutex . Wait;
   if  work_in_Q  = =  0    //  must wait!
   {    waiting_4_work  =  waiting_4_work + 1;
        wait_and_signal ( want_work,  mutex );
   }
   Packet_Q . Remove( P );  // remove from Q
   work_in_Q  =  work_in_Q  –  1;
              // (continued on next slide … )
```

Monitored_Remove **(var P : packet_buffer) con't**

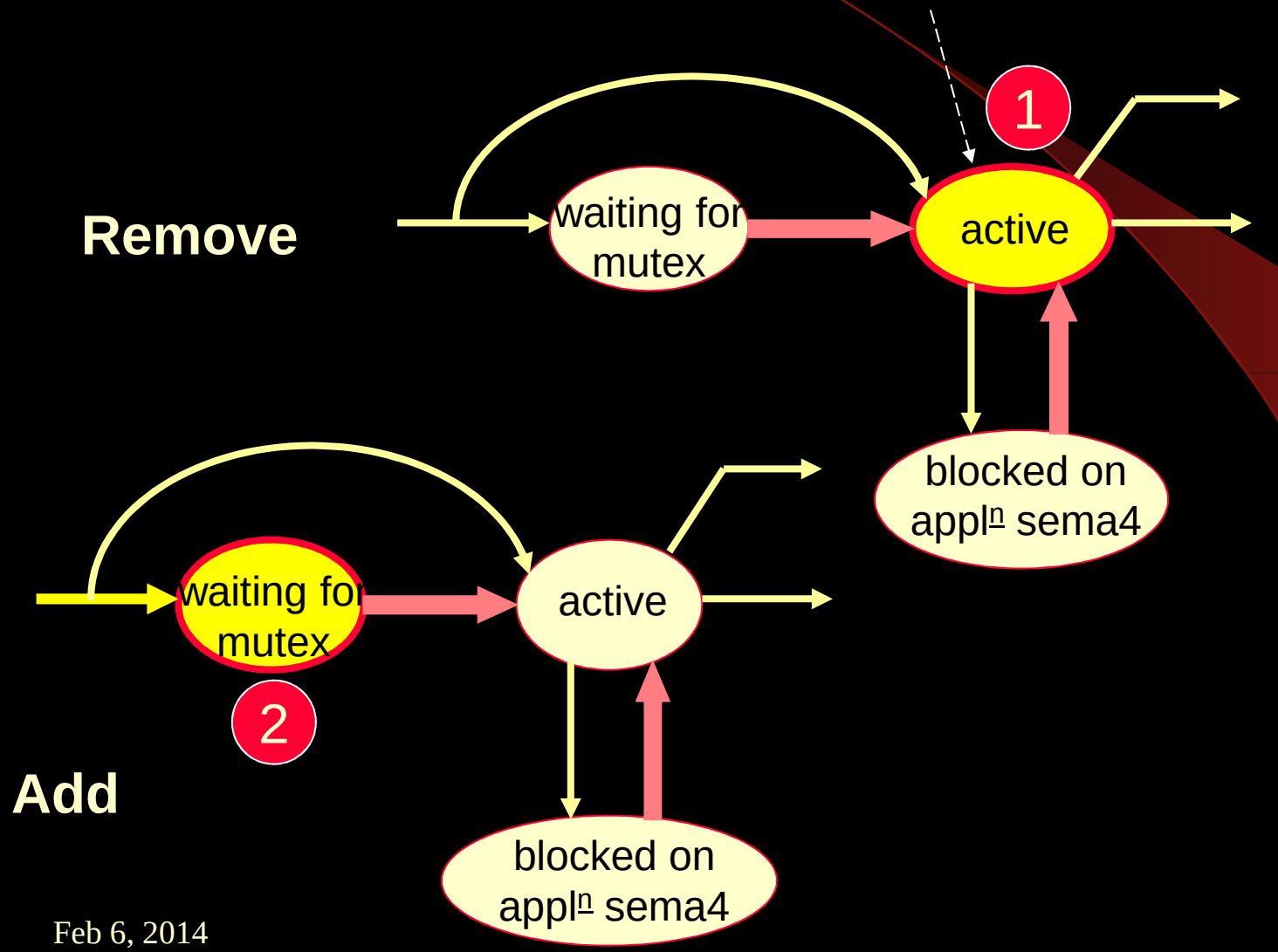
```
if  waiting_4_space  >  0   // process waiting?
{    waiting_4_space  =  waiting_4_space  –  1;
     want_space . Signal ;
     // release process waiting for space – mutex!?
} else
     { mutex . Signal }
}
```

KEY

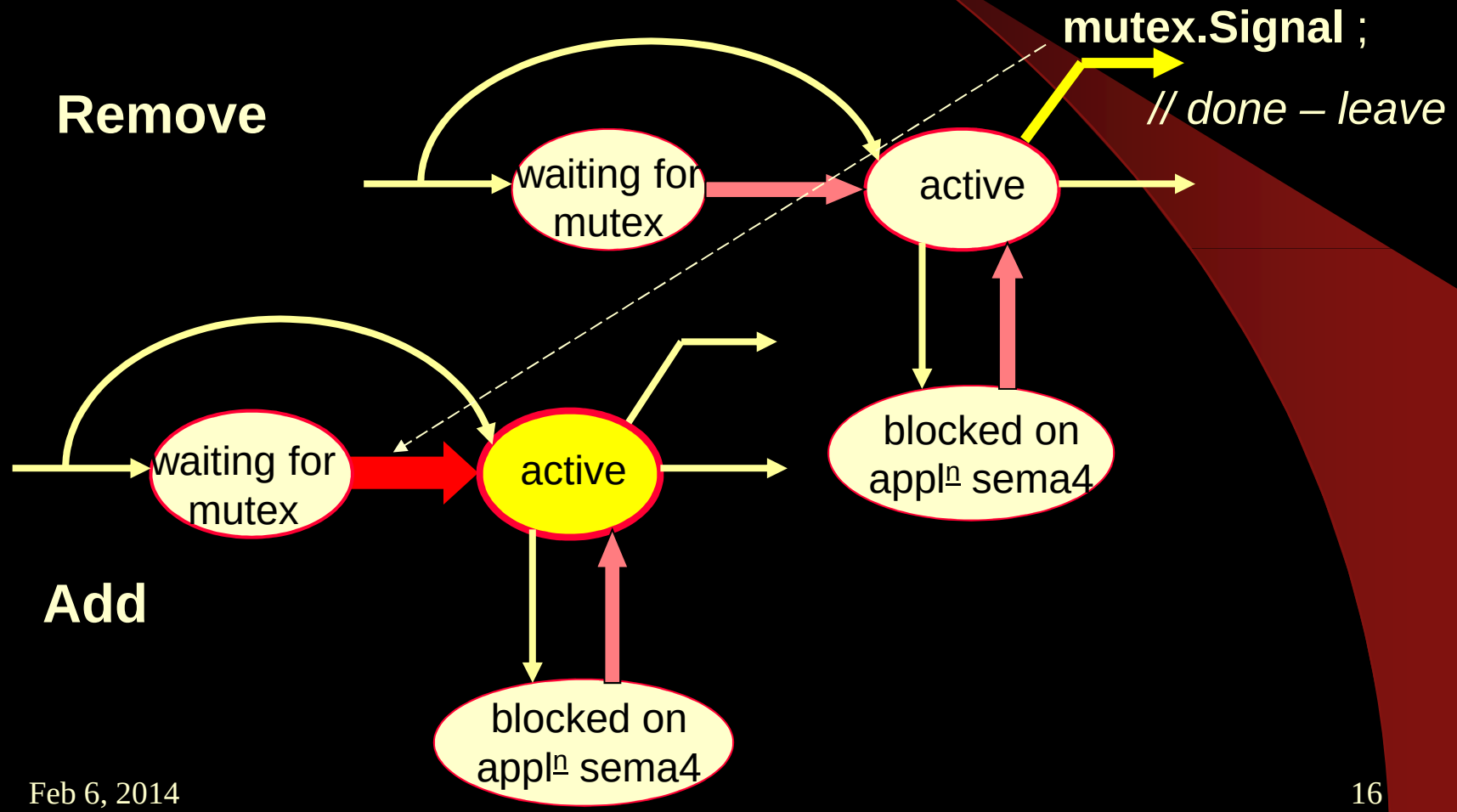
Non-Blocking Scenario

- packets in Packet_Q, but Packet_Q not full
- can Add or Remove
  - no need to block while in monitor
- suppose Remove in process 1
- new Add request is blocked at **mutex** 2
- complete scenario involves **mutex** only

Remove is Active Process



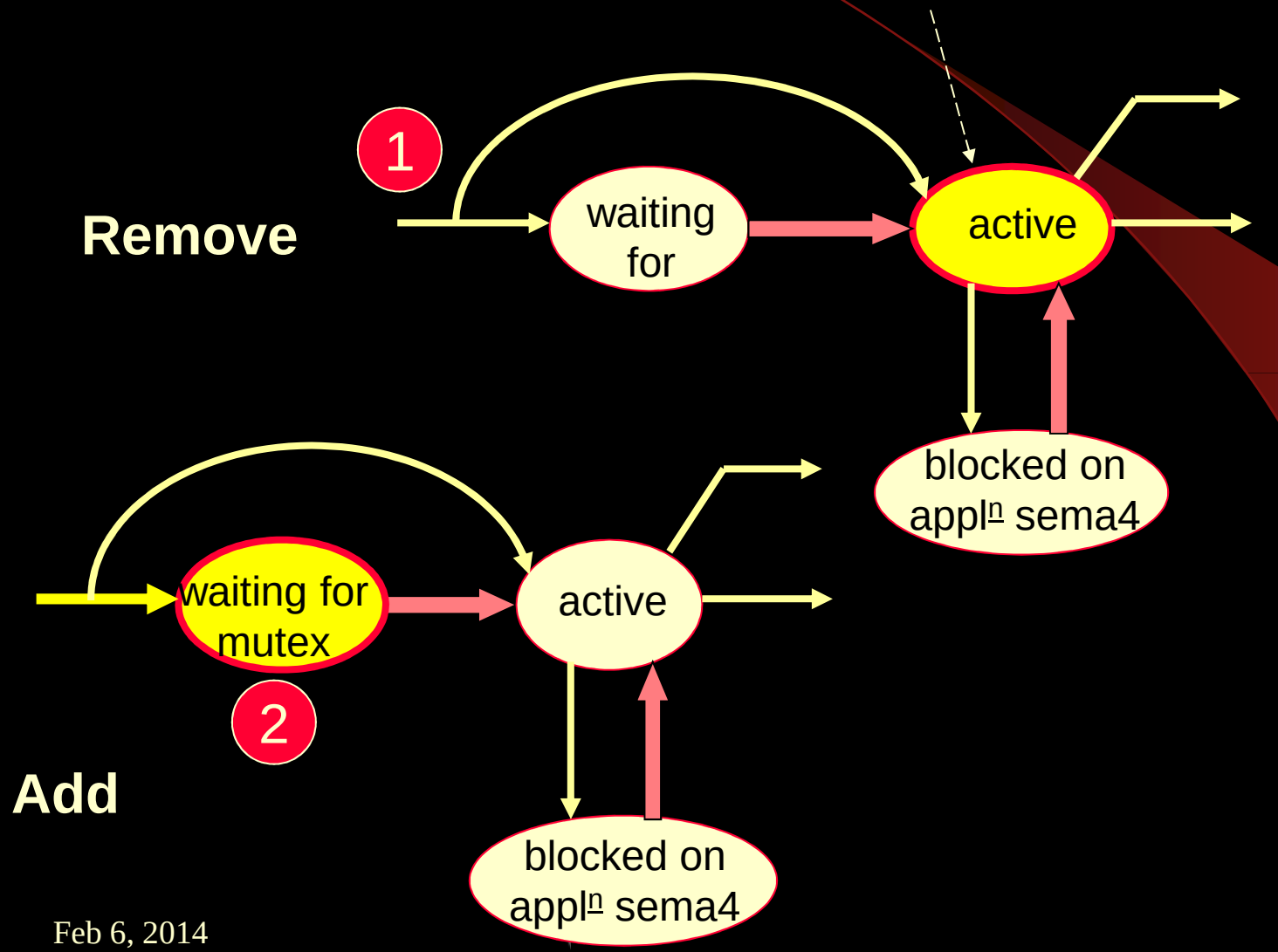
Remove Done – Release Add



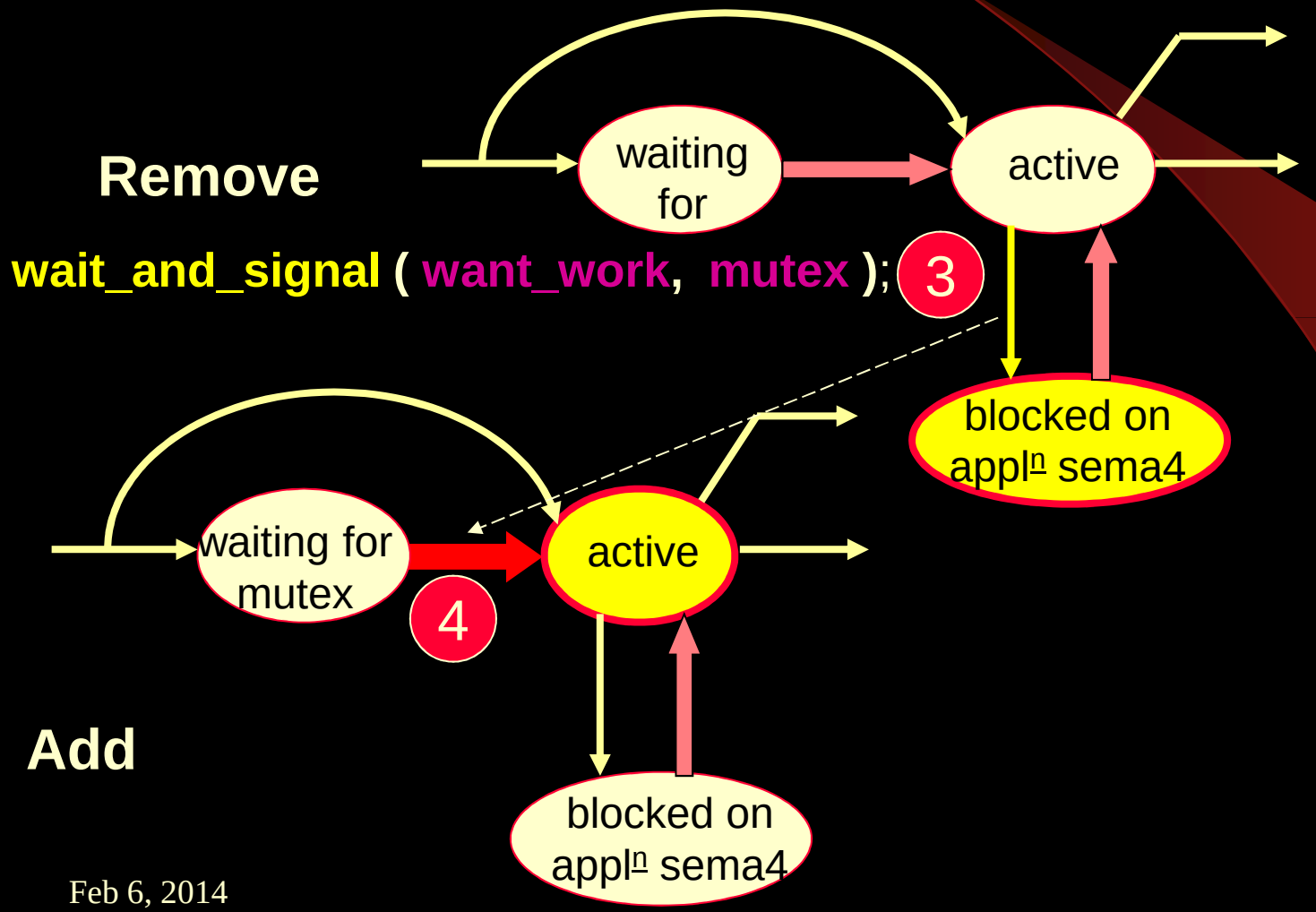
Blocking Scenario

- Packet_Q empty
- Remove begins → active 1
- Add begins – initially blocked 2
- Remove – nothing to get
  → blocks and releases Add 3 4
- Add – enqueues packet **and** releases Remove 5 6

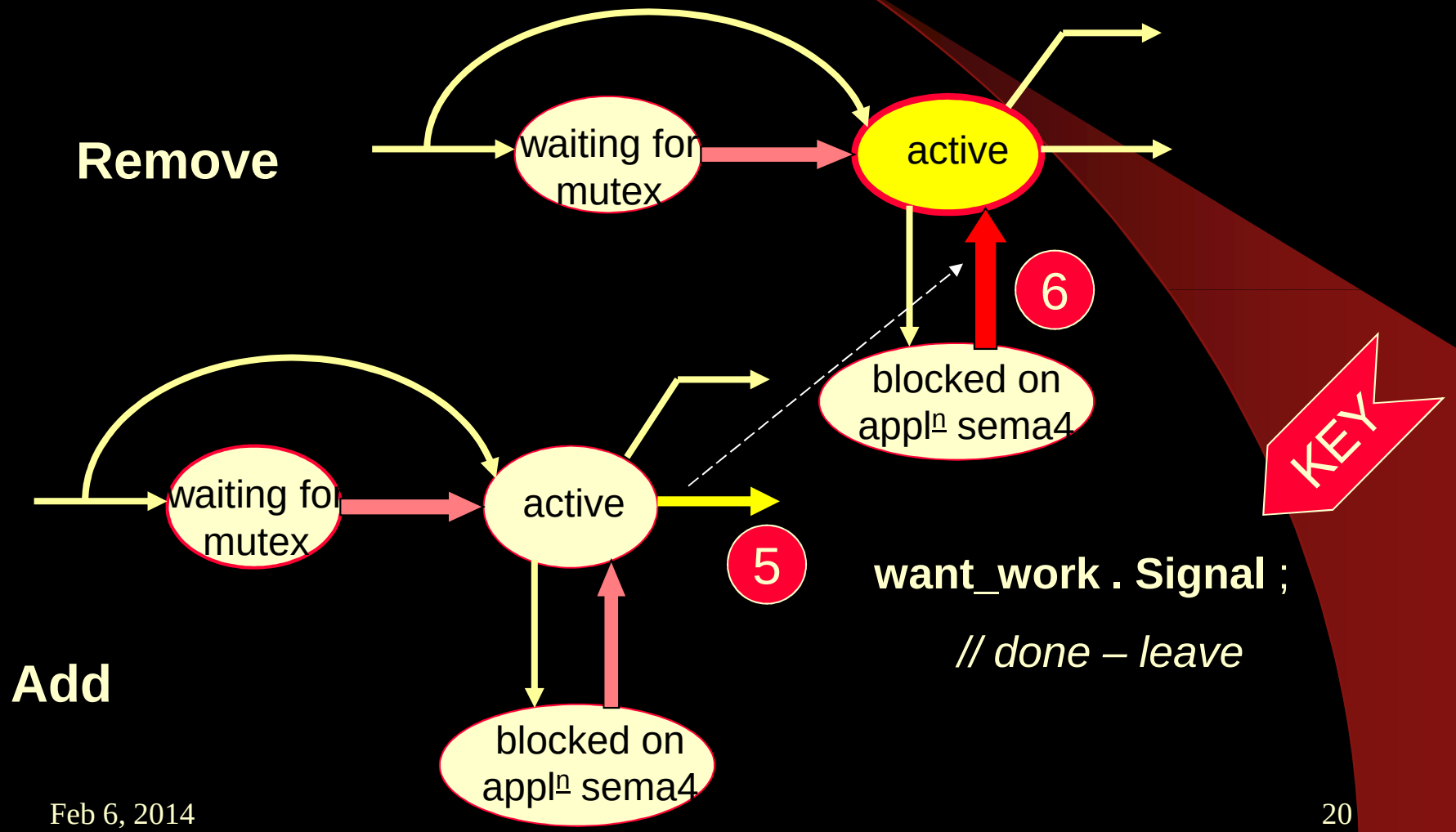
Only One Active Process



Still Only One Active Process



Remove “Owns” Mutex



Limitations in this Style of Monitor

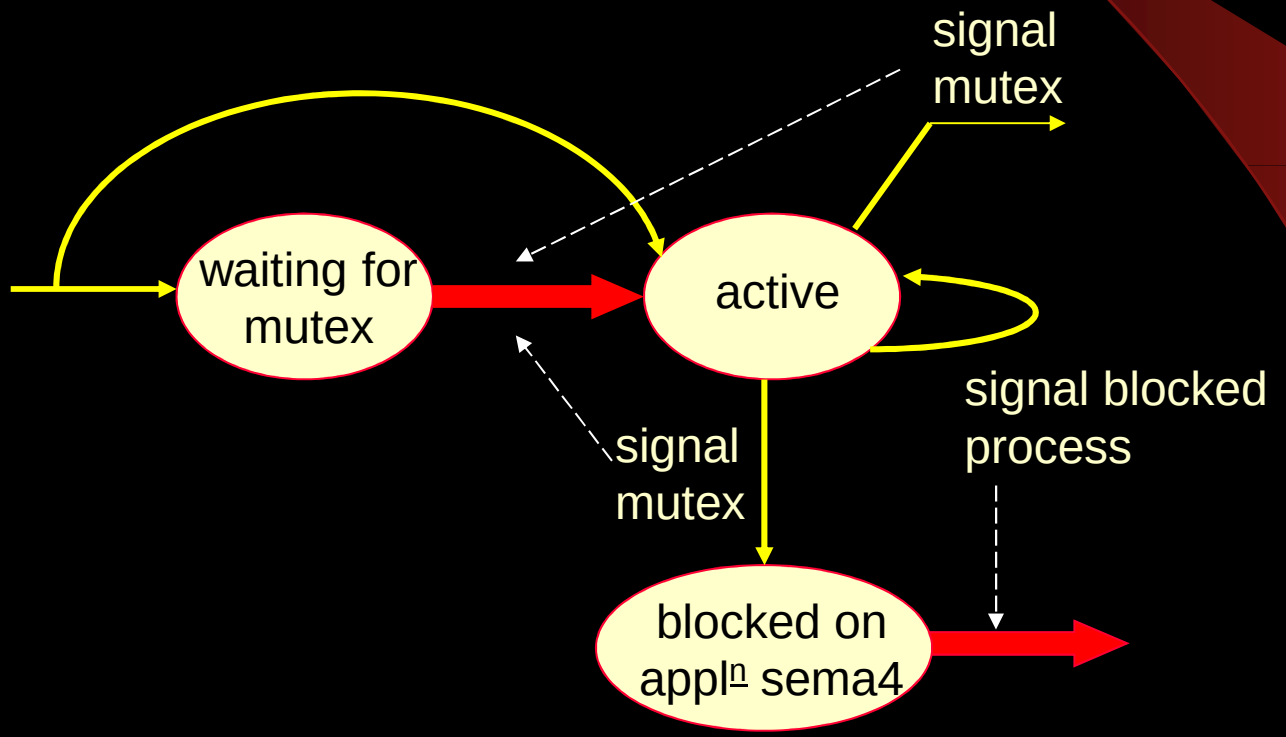
- active process can only release (at most) one process blocked on an appl^{n} sema4
- active process must leave monitor after releasing a blocked process
- too simplistic ?

mutex constraint!

Manager-Style Monitor

- release many blocked processes and stay active ☺
  - process must leave as soon as unblocked ☹

mutex constraint!

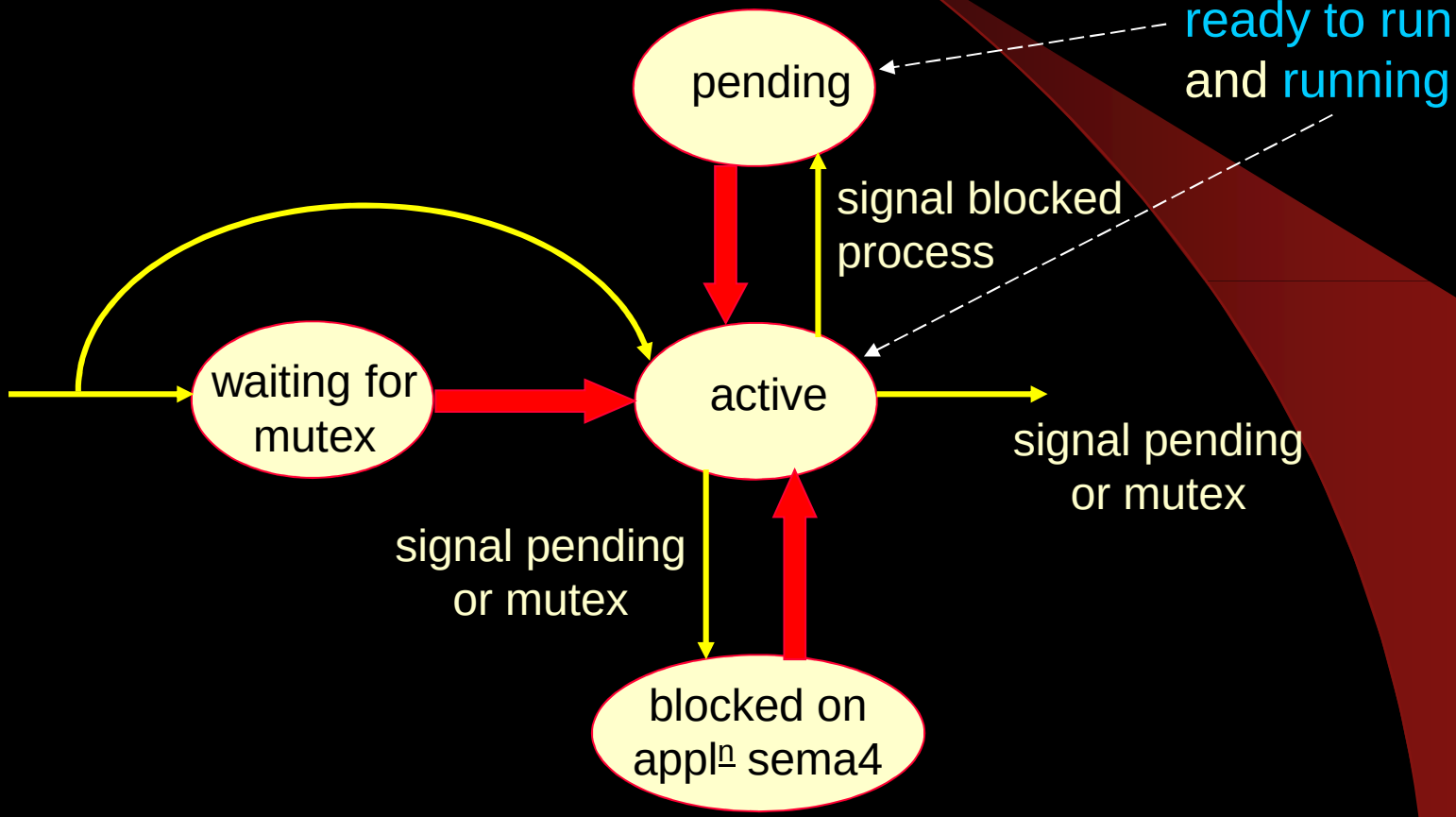


Mediator-Style Monitor

- when active process unblocks a process from an appl^{n} sema4 gives up the right to execute (but stays in monitor!)
  - Block on **"pending_sema4"**
- i.e., process executes:

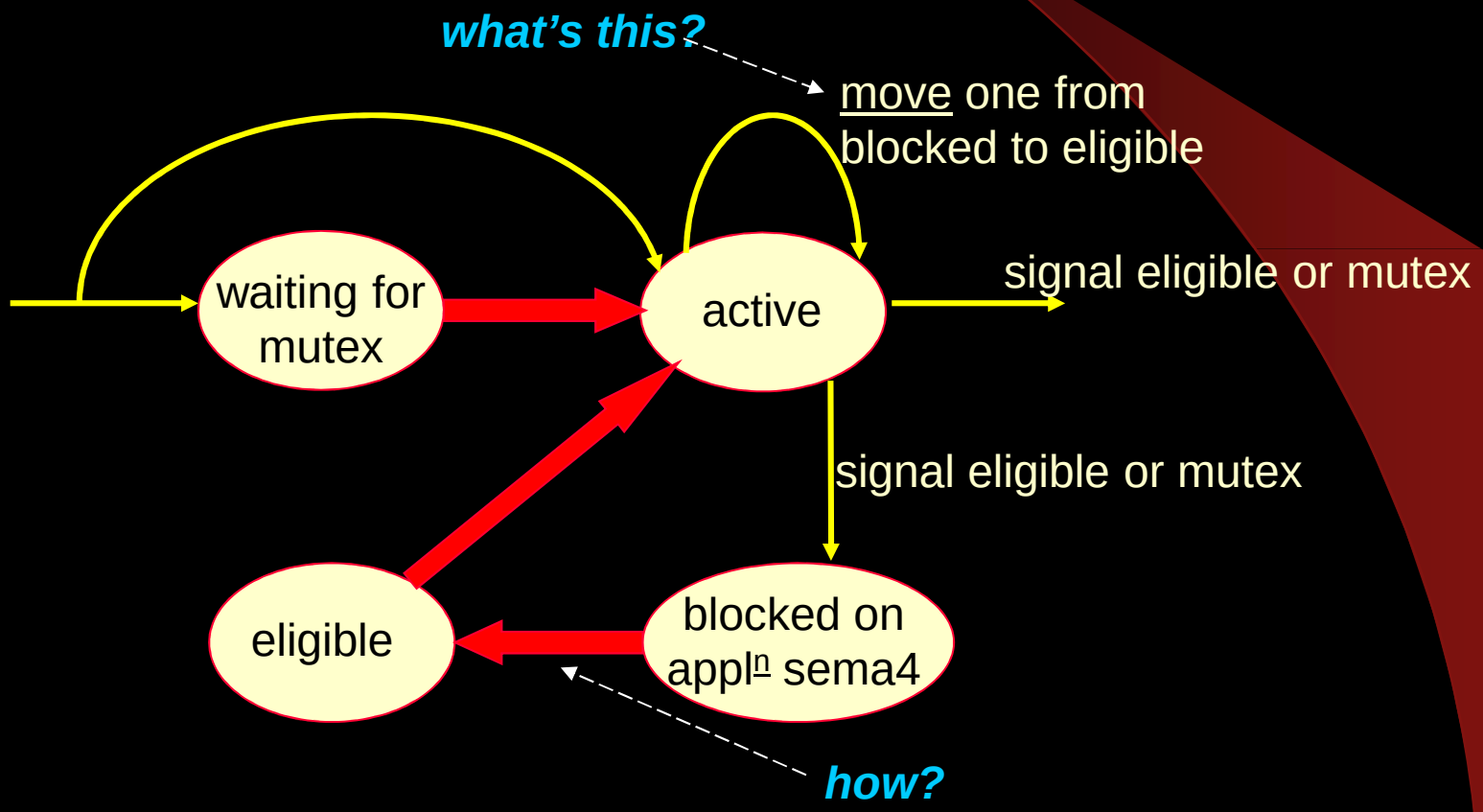
  wait_and_signal(pending_sema4, appl^{n}_sema4)
- **pending** processes are given preference over those waiting for **mutex**

Mediator


similar to
ready to run
and running
pending
signal blocked
process
waiting for
mutex
active
signal pending
or mutex
signal pending
or mutex
blocked on
appl[n] sema4

Gladiator-Style Monitor

- manages processes similar to kernel !?!?!



Gladiator

- How is this different from Mediator?
- hmm … might be sort of complicated for an average programmer to implement, but …. might be a good model for a thread manager?? ☺

Different from kernel?

No concern for Interrupts!

Issue in Gladiator

how to "**move**" a process from blocked on an appl^{n} sema4 to blocked on eligible sema4?

- sema4-to-sema4 transfer operation? nope! ☹
- unblock process to run briefly and move itself?
  - 2 active processes in monitor? ☹
    - do a context switch then run only long enough to block again (another context switch!)
    - seems like a waste of overhead! ☹

Resolving Gladiator Issue

monitor could do some process management

- each process in monitor has associated **record**
- contains at least:
  - id **of "own" sema4** (unique for each process)
  - Plus: process id? priority? applⁿ info?
- process record could be created as a local variable (in process' stack) when process enters monitor
  - process always has access to it

Queues of Process Record Ptr's

- monitor maintains queues of process record ptr's
- when active process wants to block itself and release another process:
  1. puts own record ptr in an appropriate queue
  2. decides what process to release – gets process record ptr from queue – now can access the "own" sema4 of the process to be released

wait_and_signal("own" sema4, // block itself

sema4 from step 2) // release chosen process

Move Record Ptr vs. Run

- processes block on their “own” sema4s
- monitor code decides when to release them
- can move process records among “blocking” queues without having process run !!!
- selection of process to release can include info stored in process records

Gladiator Monitor Skeleton Code

create **OwnRecord** – includes: **OwnSema4**

// enter protected section:

mutex . Wait

// active:

do some processing

decide to block in BlockedQ

put **pointer to OwnRecord in BlockedQ**

Skeleton Code con't

```
decide which process to run
    if   ! ( EligibleQ . empty )
            {  dequeue NextP from EligibleQ
               wait_and_signal (  OwnSema4,
                                  NextP → OwnSema4 )
            }  else   // EligibleQ is empty
            {  wait_and_signal ( OwnSema4, mutex) }

    etc …  do this after becoming unblocked   ☺
    What to do when leaving monitor?
```

Another Solution (Gladiator)

Suppose the kernel supports the notion of a "Sleeping" process:

- while sleeping, process is not eligible to run
- sleeping process is not in a blocking queue
- simpler than sema4 mechanism
- easy to implement:
  - **sleeping** = new process state in kernel
  - when process is "awakened", it is ready to run

Sleep Services

sleep_and_signal(sema4)

puts calling process to sleep and signals the specified sema4

sleep_and_awaken(process_id)

puts calling process to sleep and awakens the specified process

myID( )

returns process ID of caller

Revised Gladiator Using Sleeping

create **OwnRecord** – includes **process' ID**

// enter protected section:

mutex . Wait

// active:

do some processing

decide to block in BlockedQ

put **pointer to OwnRecord in BlockedQ**

Revised Skeleton Code con't

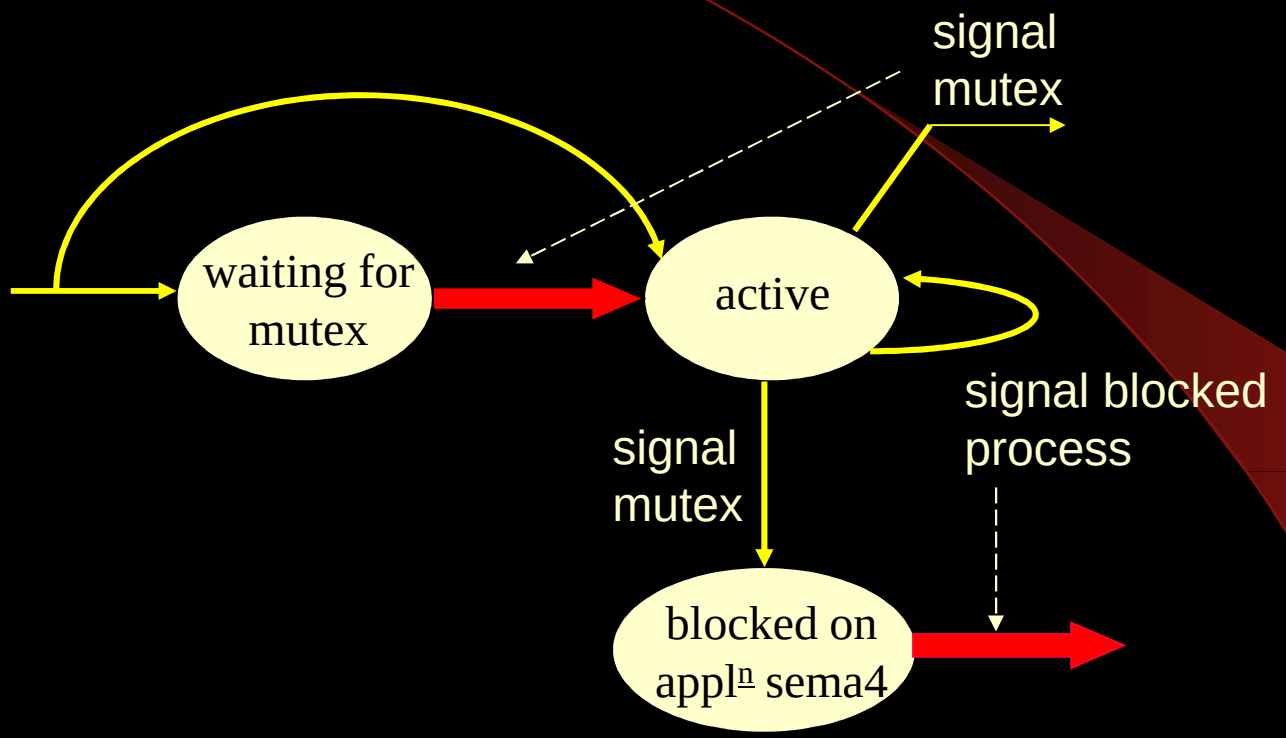
```
decide which process to run
        if   ! ( EligibleQ . empty )
                { dequeue NextP from EligibleQ
                   sleep_and_awaken ( NextP → ProcessID )
                } else   // EligibleQ is empty
                { sleep_and_signal ( mutex ) }
etc …  do this after becoming unblocked   ☺
```

less kernel overhead – "**sleeping**" is more efficient than semaphore "**blocking**"

Example: Manager-Style Timed Resource Monitor

- allow processes to request a resource
- resource is allocated based on process priority
- processes specify a maximum waiting time
- if resource is obtained within specified time, then release process with "success" return-code
- if resource not available in time, then release process with "timeout" return-code

Recall: Manager-Style Monitor



- release many blocked processes and stay active ☺
  - process must leave as soon as unblocked ☹

Kernel Support

Need **kernel services**:

- **myPriority( )**

  returns priority of calling process

- **awaken**(process_id)

  awakens the specified process

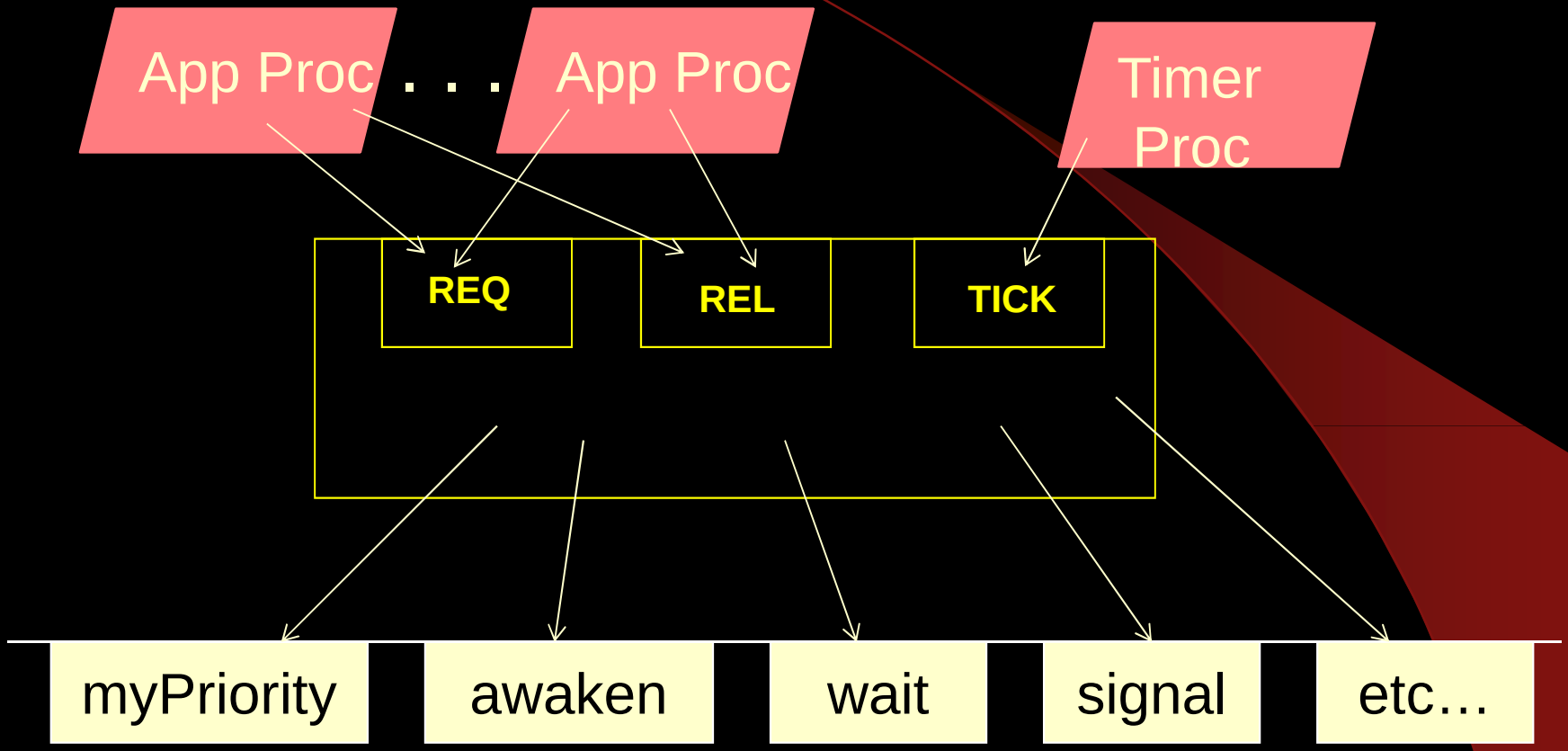
Monitor Entry Procedures:

REQ(maxTime : integer; var rtnCode : enum)
request resource: specify max. wait time in “ticks”

REL release resource

TICK called once every “tick” by a timer process (application driver … accesses mutex sema4 & may awaken processes)

Resource Monitor



kernel services

Internal Issues

- must manage waiting processes
  - sema4's won't do!
  - priority vs. FIFO wait
  - timeout release in arbitrary order

Solution: maintain **Waiting** list
  - list of processes in descending-priority
  - highest priority first

Each Process has ProcRec record

- **priority** : integer; // processes priority
- **id** : run-time_id; // processes id
- **tickslеft** : integer; // time left to wait
- var **result** : enum; // ptr to return code variable
- **next** : ProcRecPtr; // used for list management

Monitor's Persistent Variables

Available : boolean ;

// true iff resource is available

// initial value? true? false until first REL?

Waiting : ProcRecPtr = NULL;

// ptr to Waiting list

Mutex : sema4 = 1; // mutual exclusion

Monitor Code: REQ

```
REQ( maxTime : integer;  var  rtnCode : enum )
{  MyProcRec : ProcRec;  // local var
   Mutex . Wait;   // gain mutex
   if  Available        // easy – allocate immediately!
   {     Available = false;
         rtnCode = success;
         Mutex . Signal;
   }  // DONE! (easy case)
```

Wait Case

```
else  // not Available: must wait for resource
   {    // initialize ProcRec for waiting
        ProcRec . priority = myPriority( );
        ProcRec . id = myID( );
        ProcRec . tickslеft = maxTime;
        ProcRec . result = rtnCode;  // copies ref
        ProcRec . next = NULL;
```

Wait Case con't

```
// priority insert ProcRec into Waiting list
      // code omitted ☺

// wait for resource, open mutex gate
sleep_and_signal( Mutex );
// eventually – will be awakened:
      // all done! – either obtained resource, or
      // timed out – rtnCode contains result
      //    Manager-style:  leave monitor!
}   // end of else  (wait case)
}  // end of REQ
```

Monitor Code: REL

```
REL      // no param's

{  P : ProcRecPtr;  // local var
   Mutex . Wait;    // gain mutex
   if  Waiting == NULL  // none waiting – easy!
        {  Available = true; }
```

Awaken Case (in REL)

```
else  // awaken from front of Waiting list
      { P = dequeued ptr from Waiting list;
        P → result = success;     // allocate resource!
        awaken( P → id );
        //  Available remains false!
      }
   Mutex . Signal;
 } // end of REL
```

Monitor Code: TICK

```
TICK
{  Cur : ProcRecPtr;                    // local var
   Mutex . Wait;   // gain mutex
   // traverse Waiting list – manage timouts
   for (Cur = Waiting;  Cur = Cur → next;  Cur != NULL )
   {    Cur → ticksleft  =  Cur → ticksleft  –  1;
```

Time-Out Case

```
        if  Cur → tickslеft = = 0
        { // remove timed-out process from Waiting list
          // code omited ☺
          Cur → result = timeout;
          awaken( Cur → id );   // but stay Active
        }
    }  // end for loop
    Mutex . Signal;
}
```