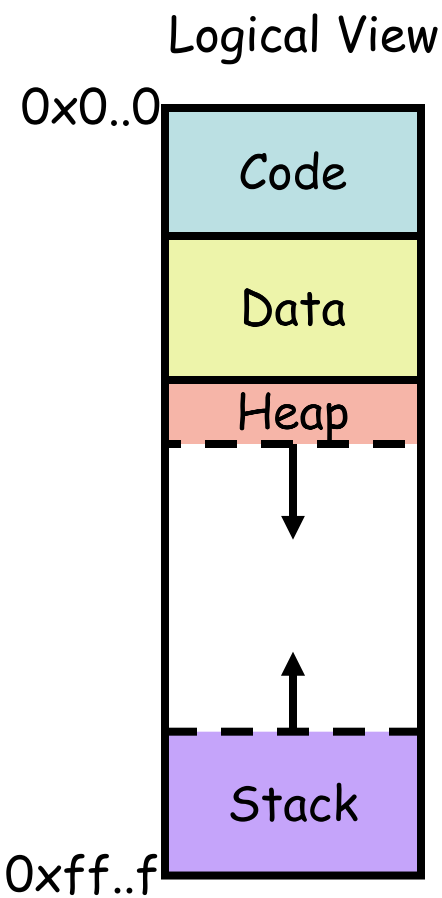
Processes and Threads

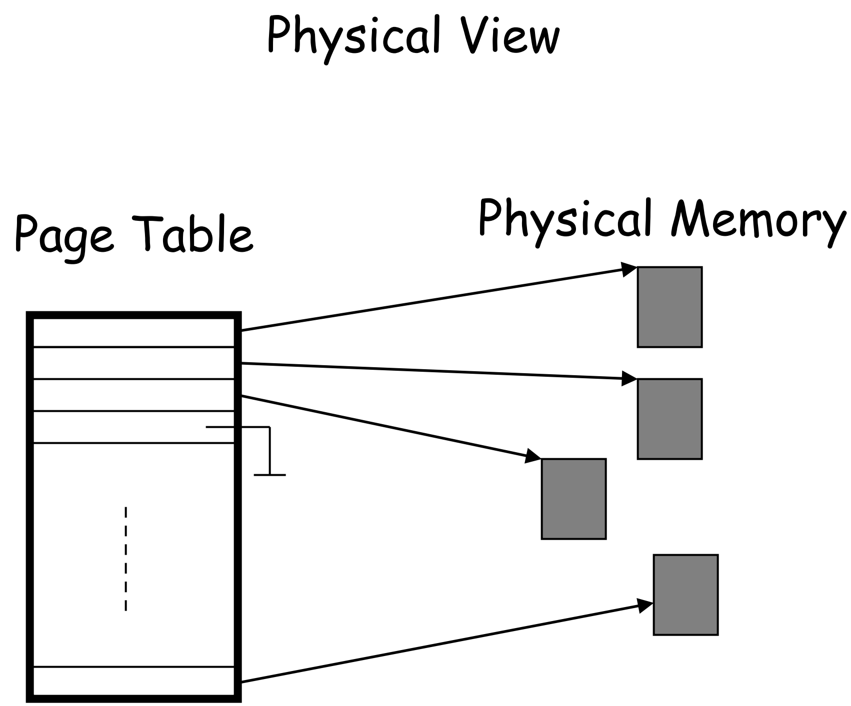
Components/Context of a Process

- Address Space
  - Code, Data, Stack, Heap
- Registers + SP + PC
- OS information: Open files, accounting info, …

- Page table of this process tracks address space
- OS keeps other information in a Process Control Block (PCB)

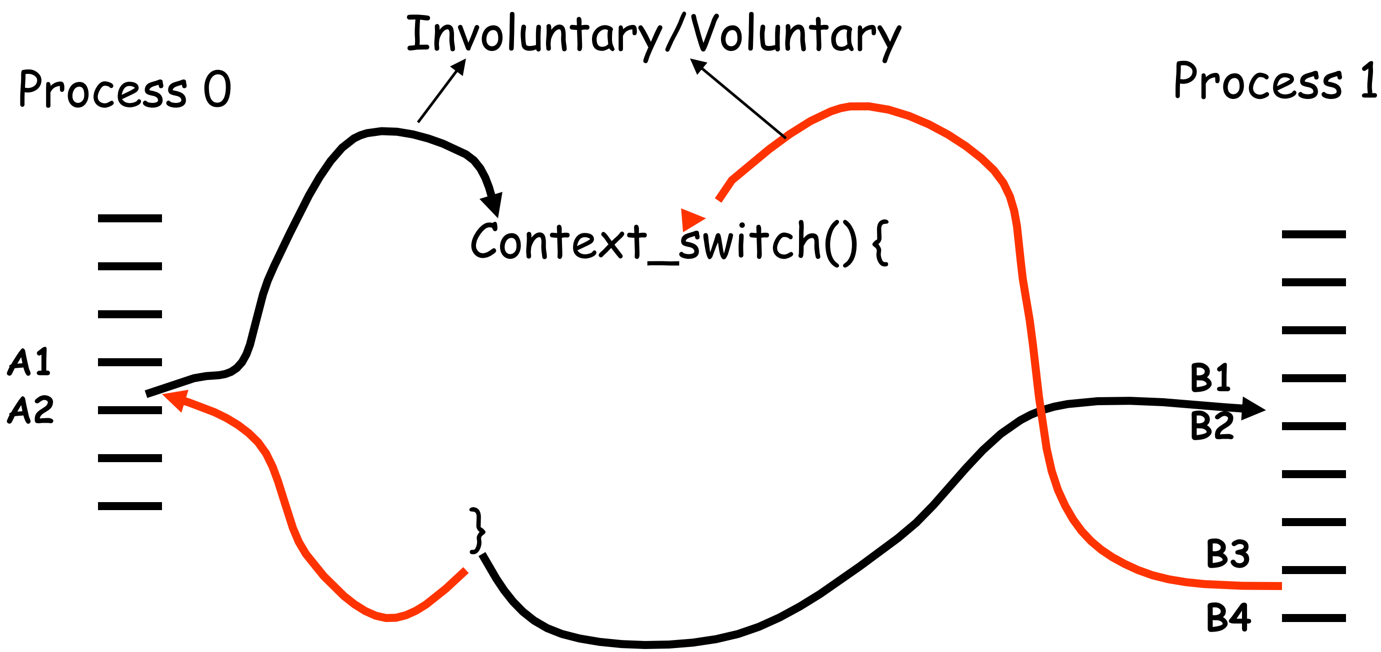
Address Space





Context Switch

To improve efficiency: Keep contexts of several processes in memory and switch periodically between them.



```
Context_switch() {

   push R0, R1, …              // save regs on its stack
   PCB[curr].SP = SP           // save stack pointer
   PCB[curr].PT = PT           // save ptr(s) to address space
   // NOTE: no point saving PC!

   next =  schedule()          // find next process to run

   PT = PCB[next].PT
   SP = PCB[next].SP
   pop Rn, … R0

   return                      // NOTE: Ctrl returns to another process
}
```

Processes are convenient to …

- Implement concurrency (between users, between activities of a user, …)
- Insulate one activity from another

Processes are NOT desirable because …

- They are heavy weight – higher scheduling (context switch) costs
  - Direct costs of switching address spaces.
  - Indirect costs (e.g. TLB/cache flushes)
- State Sharing is a problem
  - System V extensions to address this problem.

Solution: Threads

- Activities within the same address space.
- Threads within a process share (code, data, heap).
- Only stacks are disjoint.
- Switching between threads only involves switching stacks.
- Sharing is implicit
- No protection between threads of a process – but this is OK since they are meant to be cooperative.

A Disk Server: Using 1 thread of control

```
Server() {
   while () {
            receive request;
            case (REQUEST) {
                     READ:
                              If device busy {
                               queue request
                              }
                              else {
                                send request to disk
                                queue for response
                              }

                     …….
                     RESPONSE_FROM_DISK:
                              check concerned queue
                              Dequeue
                              Reply back to requester
            }
   }
}
```

Cannot block!
Else you will
loose
concurrency!

A Disk Server: Using Multiple Threads

```
Server() {
  while () {
          receive_request()
          case (REQUEST) {
                  READ:  fork_thread(read_disk, ….);
                  WRITE: fork_thread(write_disk,…);
                  …
          }
  }
}

Read_disk() {
  While (device busy)
          ;
   Program disk
   While (results not available)
          ;
   Reply back to requester
}
```

Can afford to block without loss in throughput

Advantages of threads

- Cheaper to switch than processes
- Easier to program (do not worry about blocking)
- Easier to communicate (memory is implicitly shared)

Threads can be implemented on

(a) Uniprocessors
(b) Symmetric Multiprocessors (SMPs)
(c) Cache coherent NUMA Multiprocessors (CC-NUMA)
(d) Message passing multiprocessors
(e) Networked/distributed workstations

- Efficiency decreases from (b) to (e)

Flavors of threads

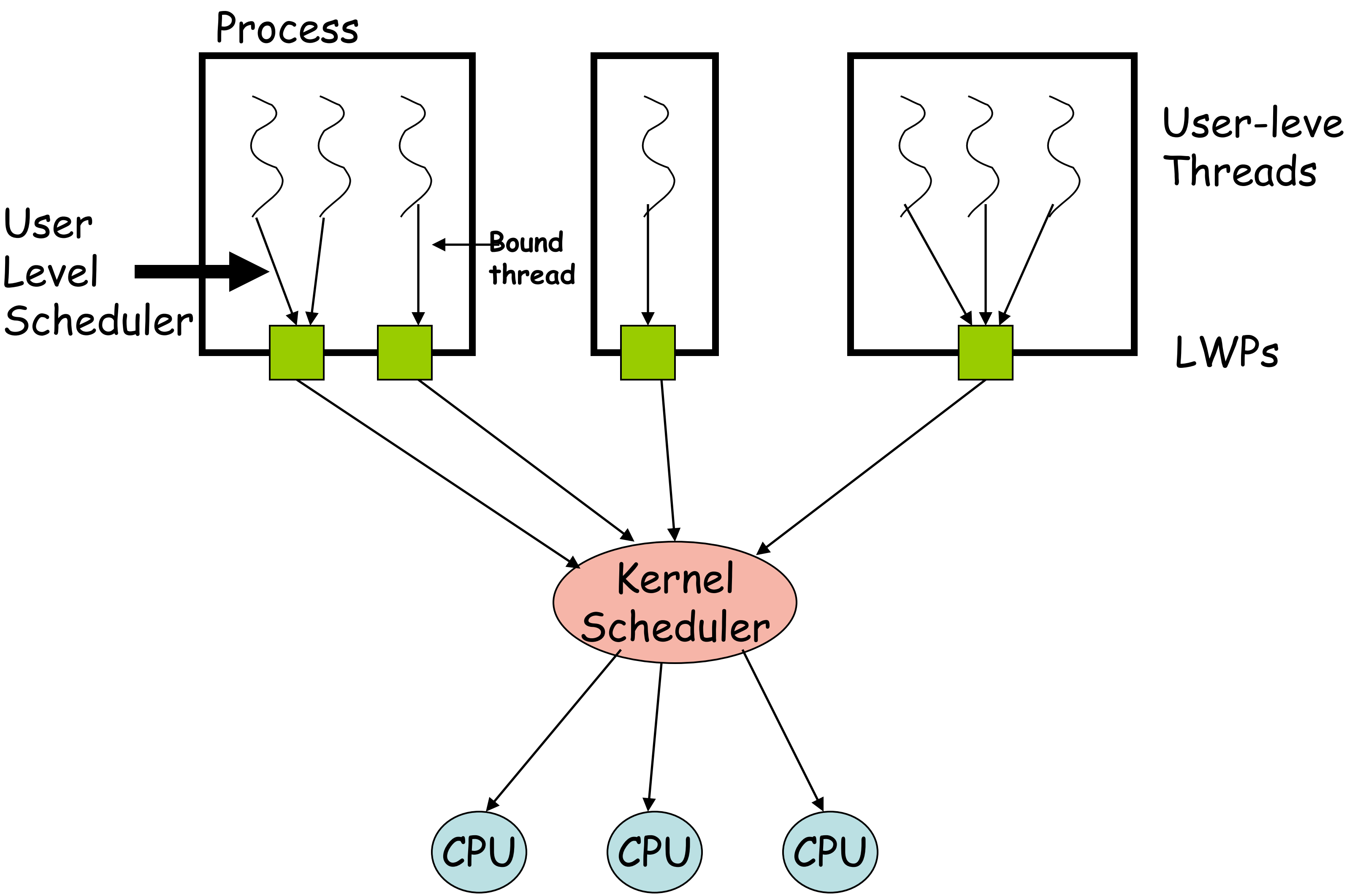
- Entirely user-level threads
  - Threads are created and managed entirely by the user code/libraries
  - Kernel is unaware of their existence
  - Advantages: Scheduling/switching can be more efficient
- Kernel-level threads
  - Kernel is the one creating (by explicit user calls) and managing the threads
  - Advantages: Better resource allocation across address spaces.

Solaris Multithreaded Architecture

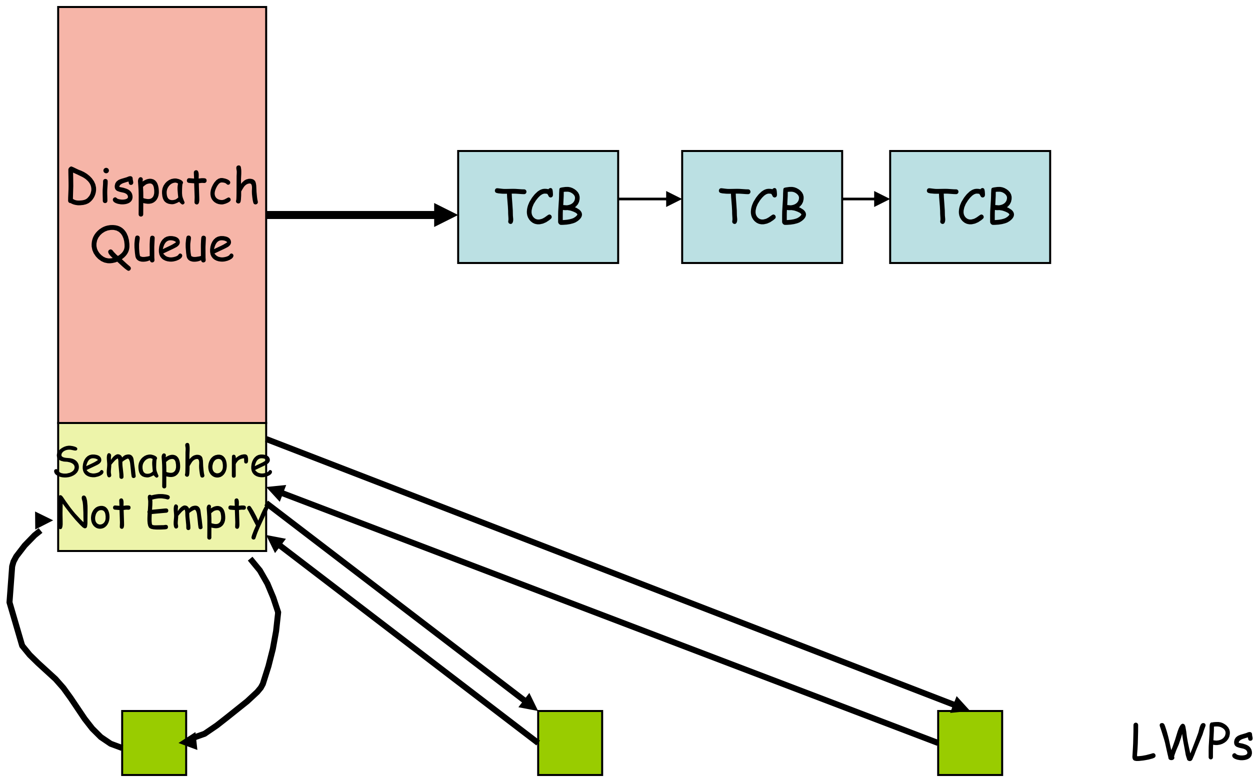
- Both user and kernel level threads
- Kernel level threads are synonymous with Light Weight Processes (LWPs)
- In addition, user can create user-level threads that will be run by these LWPs

- User is provided with processes
- User can create user-level threads via a user-level library (e.g. pthreads)
- User can specify how many LWPs should run these user-level threads
- User can bind threads to LWPs if needed or can dynamically schedule the threads to LWPs (via the library).
- Kernel schedules the LWPs on the available CPUs.

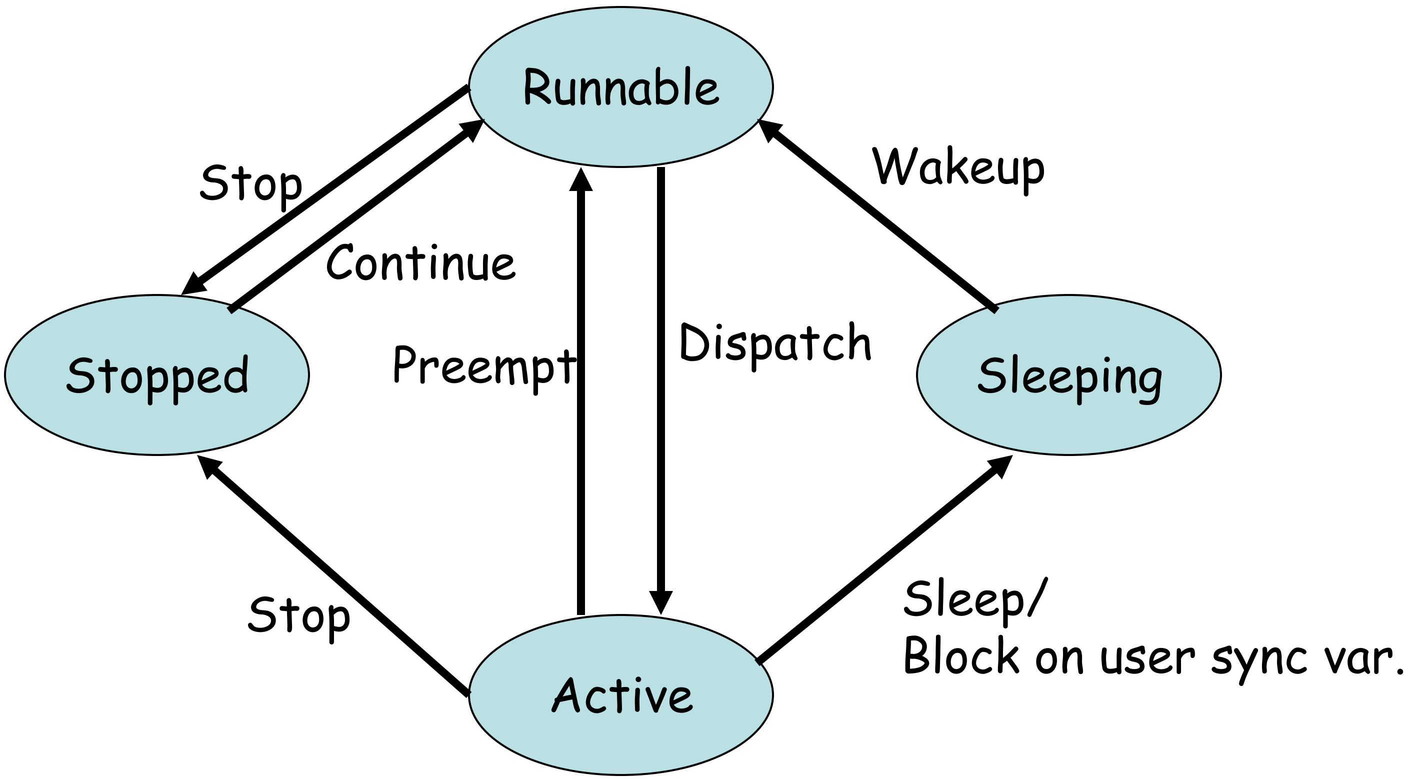
Solaris Architecture



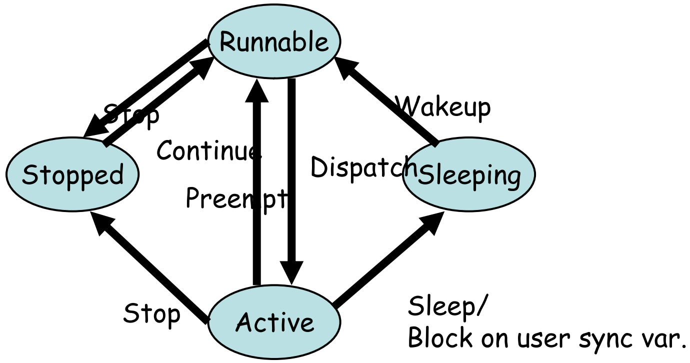
User-level scheduling

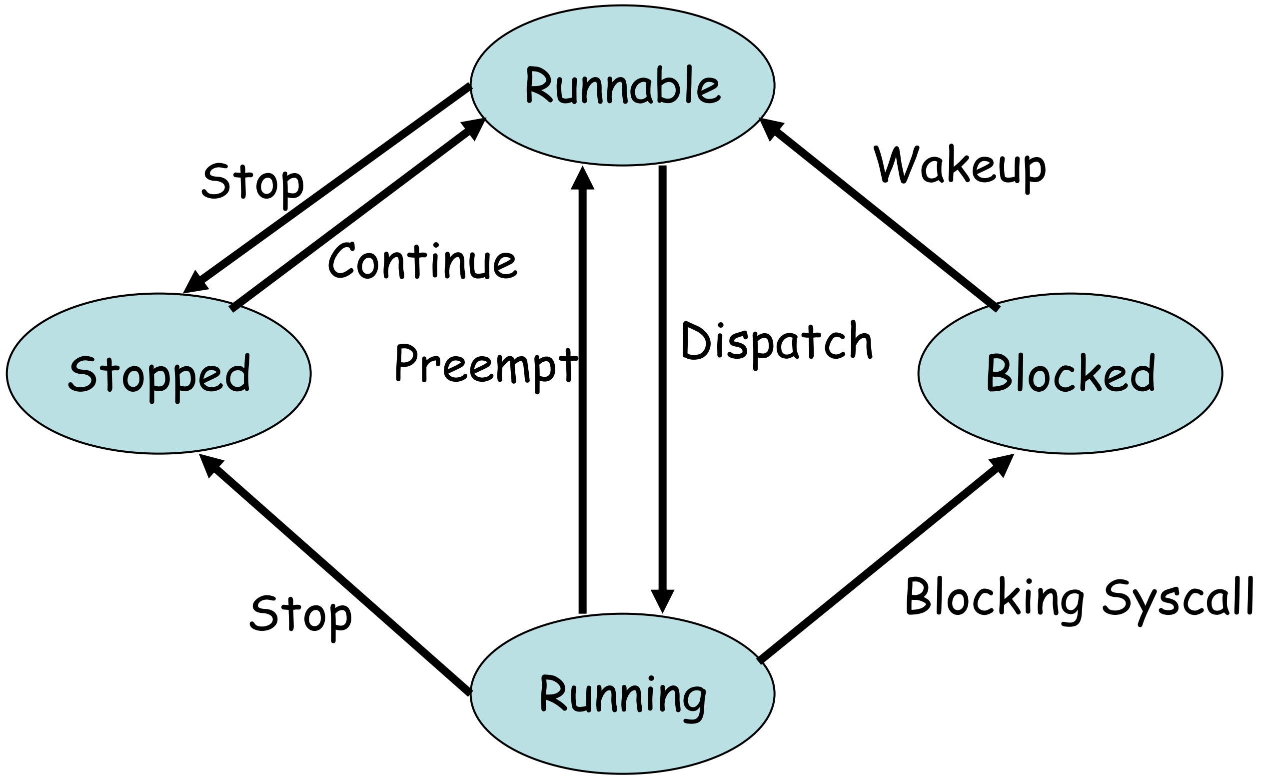


User-level thread States



LWP States


Runnable
Stop
Continue
Preempt
Dispatch
Wakeup
Stopped
Sleeping
Stop
Active
Sleep/
Block on user sync var.


Runnable
Stop
Continue
Preempt
Dispatch
Wakeup
Stopped
Blocked
Stop
Running
Blocking Syscall

SIGWAITING

- When all LWPs of a process are blocked in the kernel, the process is sent a SIGWAITING signal.
- You can choose to add create LWPs if needed
- This is a way of avoiding the problem of the kernel being ignorant of the workload within the process.

User-level scheduling

- Can be pre-emptive or not
- You can implement your own scheduler

Scheduling LWPs

- 3 scheduling classes
  - Timesharing (TS)
    - For user processes
    - Time slices the CPU
    - 0-59 priority levels
  - System
    - For some kernel activity (interrupts, etc.)
    - Priority levels 60-99
    - Usually fixed priority
  - Real-time
    - Very time sensitive
    - Fixed (higher) priorities 100-159

Threads Interface

- Thread creation, termination
- Communication (implicitly shared memory)
- Synchronization (mutex, cond variables, semaphores, etc.)

Thread Control

- t = t_create(func, arg, flags): Creates a thread that would start executing func()
- t_exit(): thread is done
- t_join(t) waits for the specified thread to exit
- t_set_prio()/t_get_prio(): set/get thread priorities
- t_yield(): voluntarily relinquishing CPU

Synchronization

- Locks
- Condition Variables
- Semaphores
- Monitors

- What determines your choice of which sync mechanism to use?

Locks

- Used to guard sections of code where shared data is manipulated
- Ordering is not as important (only exclusion)
- Waiting for events to happen is not as easy to implement.

- Exercise: How would you implement a lock construct (lock/unlock operations) in a threads library?

Condition Variables

- C_wait() and c_signal() operations.
- A thread blocked on c_wait() returns when another performs a c_signal().

- What differentiates a condition variable from a (boolean) semaphore?
- Signals can get lost!
  - i.e. if the signal is done before the wait, then signal is lost

```
Cond_t  not_full, not_empty;;
Int count == 0;

Append() {
        if count == N   c_wait(not_full);

        ... ADD TO BUFFER, UPDATE COUNT ...

        c_signal(not_empty);
}

Remove() {
        if count == 0       c_wait(not_empty);

        ... REMOVE FROM BUFFER, UPDATE COUNT

        c_signal(not_full);
}
```

What is wrong?

```
Cond_t  not_full, not_empty;
Mutex_lock m;
Int count == 0;

Append() {
            mutex_lock(m);
            if count == N   c_wait(not_full,m);

            … ADD TO BUFFER, UPDATE COUNT …

            c_signal(not_empty);
            mutex_unlock(m);
}

Remove() {
            mutex_lock(m);
            if count == 0   c_wait(not_empty,m);

            … REMOVE FROM BUFFER, UPDATE COUNT

            c_signal(not_full);
            mutex_unlock(m);
}
```

NOTE: You can improve this code for more concurrency!

Semaphores

- Condition variables with state (signals) preserved
- Counting semaphores offer more flexibility
- P() and V() operations.

Monitors

- Encapsulation of Shared Data Objects and operations on them (an ADT), with the property that only 1 activity can be "virtually running" (i.e. running+ready) at any time.
- Operations use synchronization so that the invoker does not need to worry about them.
- You can have wait/signal primitives within the monitor to block/proceed when needed (note they are similar to condition variables, except you do not need a lock to go with it!).

PENNSTATE

```
Monitor Bounder_Buffer {

Buffer[0..N-1];
Int count= 0, head=tail=0;
Cond_t not_full, not_empty;

Append(Data) {
        if count == N   wait(not_full);
        Buffer[head] = Data
        count++;
        head = (head+1)%N;
        if !empty(not_empty) signal(not_empty);
}

Remove() {
        if count == 0 wait(not_empty);
        Data = Buffer[tail];
        count--;
        tail = (tail+1)%N;
        if !empty(not_full) signal(not_full);
}

}
```

PENNSTATE

Monitor Readers_Writers {

```
Database [];
Int nreaders, nwriters;
Cond readq, writeq;

Read() {
        while (nwriters > 0) wait(readq);
        nreaders++;
        signal(readq);

        …. Read database …

        nreaders--;
        if (nreaders == 0) signal(writeq);
}

Write() {
        while (nreaders>0) || (nwriters>0) wait(writeq);
        nwriters++;

        … Write database

        nwriters--;
        if !empty(readq) signal(readq);
        else if !empty(writeq) signal(writeq)
}
}
```

Monitor Readers_Writers {

```
Int nreaders, nwriters;
Cond readq, writeq;

Start_Read() {
        while (nwriters > 0) wait(readq);
        nreaders++;
        signal(readq);
}

End_Read() {
        nreaders--;
        if (nreaders == 0) signal(writeq);
}

Start_Write() {
        while (nreaders>0) || (nwriters>0) wait(writeq);
        nwriters++;
}

End_Write() {
        nwriters--;
        if !empty(readq) signal(readq);
        else if !empty(writeq) signal(writeq)
}
}
```