Threads, Scheduling, and Synchronization

Dr. Daniel Andresen

CIS520 – Operating Systems

Threads vs. Processes

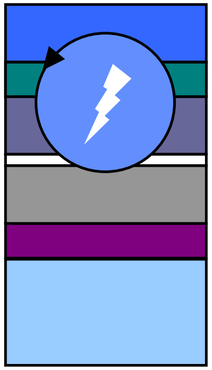
1. The *process* is a *kernel abstraction* for an independent executing program.

   includes at least one "thread of control"

   also includes a private address space (VAS)

   - requires OS kernel support

   (but some use *process* to mean what we call *thread)*



2. Threads may share an address space

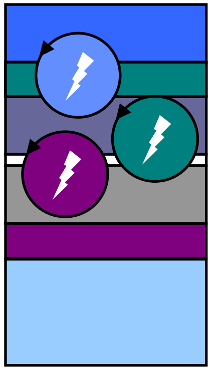
   threads have "context" just like vanilla processes

   - *thread context switch* vs. *process context switch*

   every thread must exist within some process VAS

   processes may be "multithreaded"

Thread::Fork

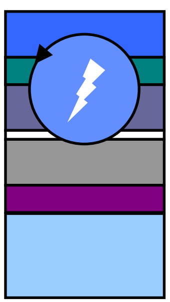


Implementing Threads in a Library

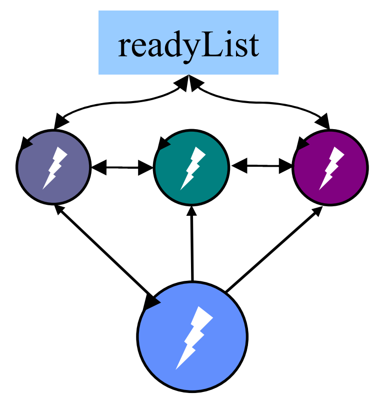
The Nachos library implements *user-level threads*.

coroutines

- no special support needed from the kernel (use any Unix)
- thread creation and context switch are fast (no syscall)
- defines its own thread model and scheduling policies



```
while(1) {
    t = get next ready thread;
    scheduler->Run(t);
}
```

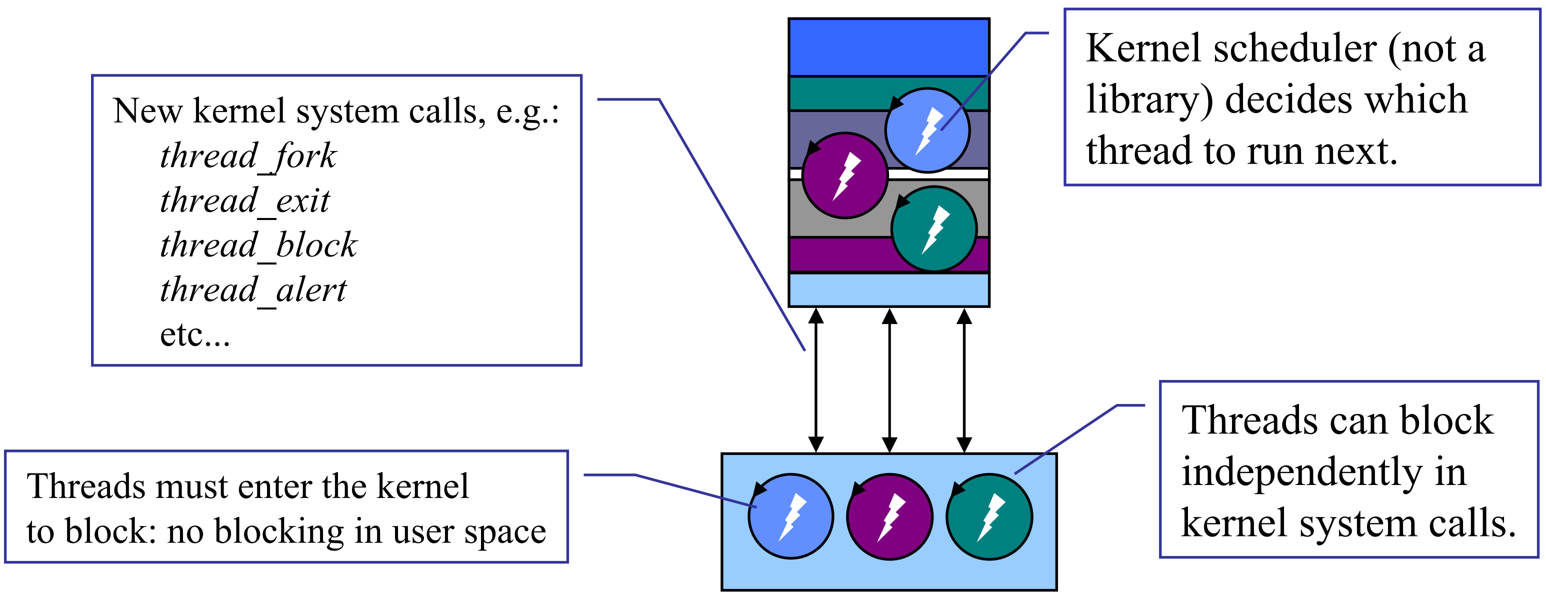


Kernel-Supported Threads

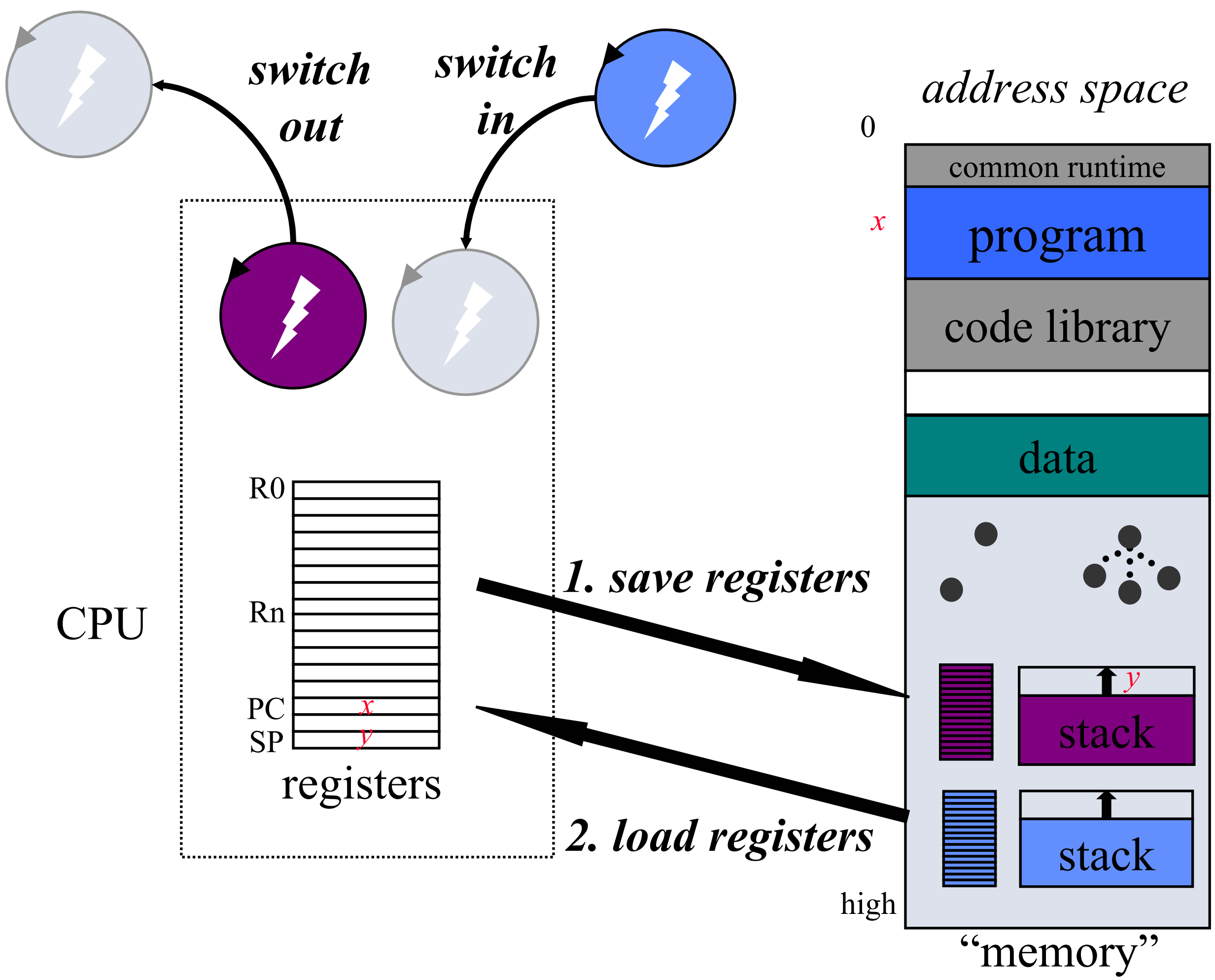
Most newer OS kernels have *kernel-supported threads*.

- thread model and scheduling defined by OS

  NT, advanced Unix, Linux, etc.



Thread Context Switch



A Nachos Context Switch

```
/*
 * Save context of the calling thread (old), restore registers of
 *  the next thread to run (new), and return in context of new.
 */
switch/MIPS (old, new) {
        old->stackTop = SP;
        save RA in old->MachineState[PC];
        save callee registers in old->MachineState

        restore callee registers from new->MachineState
        RA = new->MachineState[PC];
        SP = new->stackTop;

        return (to RA)
}
```

*Save current stack pointer and caller's return address in **old** thread object.*

Caller-saved registers (if needed) are already saved on the thread's stack.

Caller-saved regs restored automatically on return.

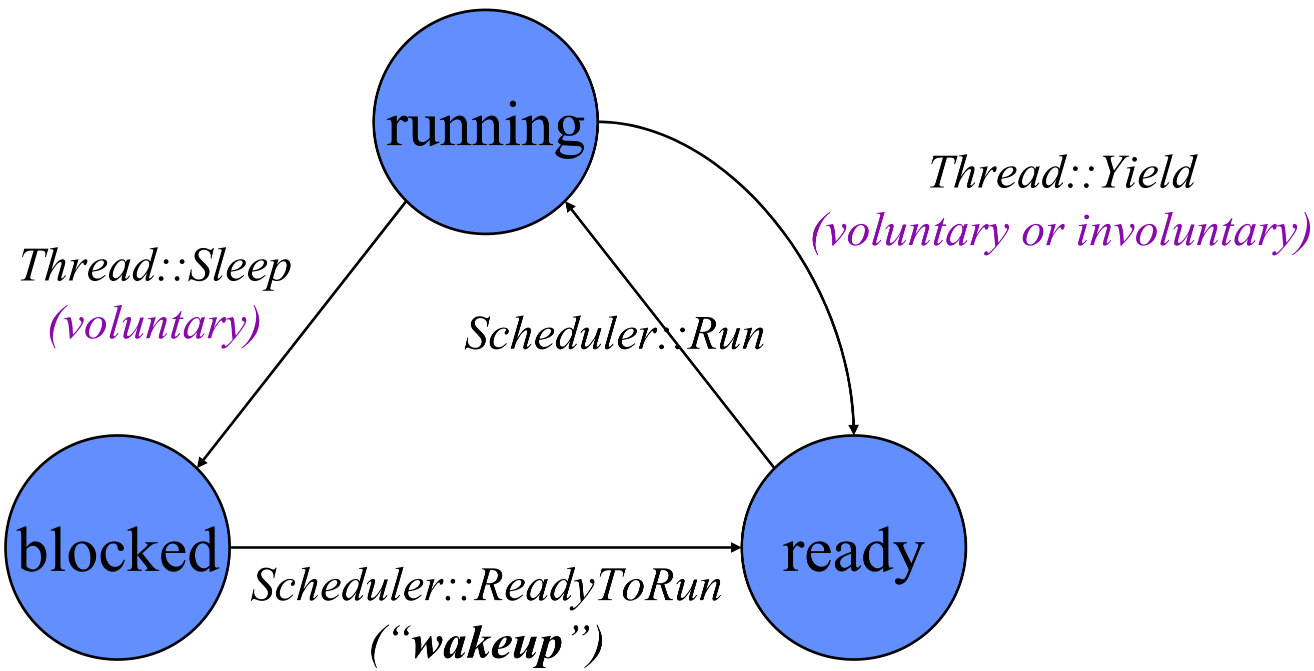
*Switch off of **old** stack and back to **new** stack.*

*Return to procedure that called switch in **new** thread.*

Blocking in *Sleep*

- An executing thread may request some resource or action that causes it to *block* or *sleep* awaiting some event.
  - passage of a specific amount of time (a ***pause*** request)
  - completion of I/O to a slow device (e.g., keyboard or disk)
  - release of some needed resource (e.g., memory)
  - In Nachos, threads block by calling ***Thread::Sleep.***
- A sleeping thread cannot run until the event occurs.
- The blocked thread is awakened when the event occurs.
  - E.g., ***Wakeup*** or Nachos ***Scheduler::ReadyToRun(Thread* t)***
- In an OS, threads or processes may sleep while executing in the kernel to handle a system call or fault.

Thread States and Transitions

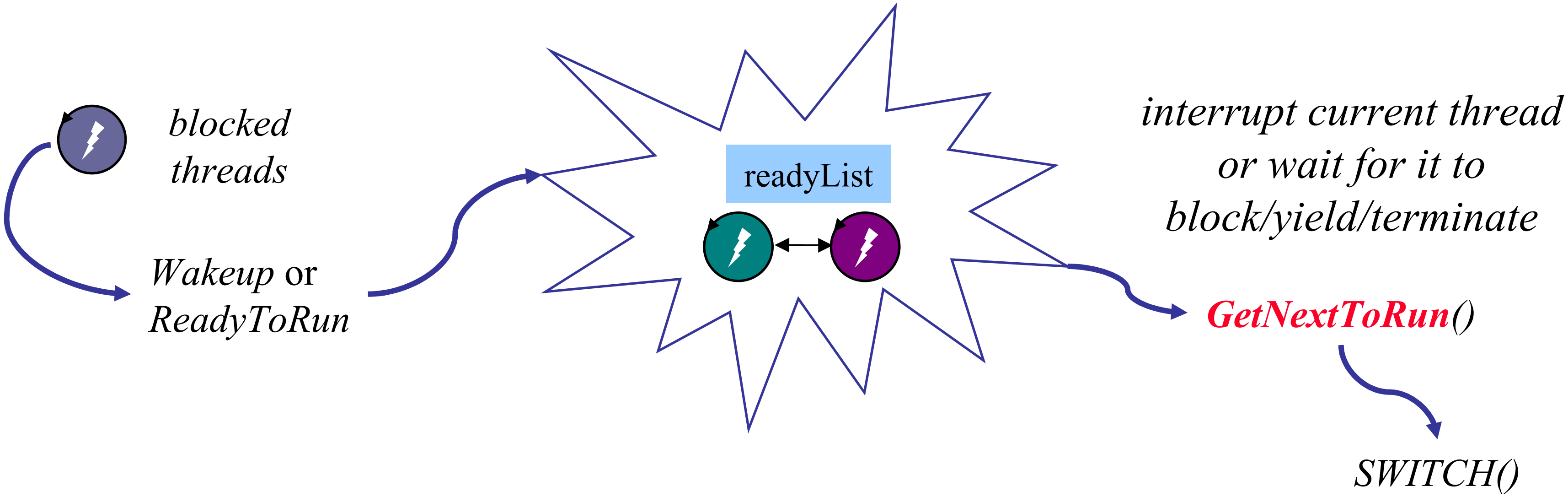


CPU Scheduling 101

The CPU scheduler makes a sequence of "moves" that determines the interleaving of threads.

- Programs use synchronization to prevent "bad moves".
- …but otherwise scheduling choices appear (to the program) to be *nondeterministic*.

The scheduler's moves are dictated by a *scheduling policy*.



Scheduling algorithms

- determines which processes use the CPU.
- Can determine overall feel of system.
- **Guiding principle**: CPU-I/O burst cycle
- **Types**: preemptive and nonpreemptive
- **Criteria**:
  - CPU Utilization
  - Throughput – completions/time period
  - Turnaround time – total execution time
  - Waiting time – time spent in ready queue
  - Response time – time until first response

First Come, First Served (FCFS)

- simple, intuitively fair
- usually awful performance, wide swings in response times, nonpreemptive
- Example: assume 3 jobs

Process	Burst time
P1	24
P2	3
P3	3

Arrival: P1, P2, P3

Avg. wait: (0 + 24 + 27)/3 = **17 ms.**

Arrival: P2, P3, P1

Avg. wait: (0 + 3 + 6)/3 = **3 ms.**

Shortest Job First (SJF)

- provably optimal
- major problem – estimating length of next CPU burst.
- Use user-supplied values? Incentive to lie…
- Could use prediction
- Fine algorithm for batch jobs, long-term scheduling.
- Tough for short-term scheduling
- Variant - Preemptive SJF (PSJF) = Shortest-Remaining-Time-First.

Round Robin (RR)

- each process is assigned a time interval, called its **quantum**, for which it is allowed to run before being interrupted.
- Usually all processes are assigned the same time quantum, *q*
- Note, as q increases, the CPU efficiency also increases, and as q decreases, the CPU efficiency also decreases.

  Let q = 20 msec. Suppose that a context-switch takes c = 5 msec. Then, 5/(20+5) = 20% of the time is wasted doing context-switches.

  CPU utilization (or efficiency) is the amount of time spent doing good work (20 msec) divided by the total time (25 msec). In this example, the cpu efficiency is .80; that is, 80%. Let q = 495 msec, and c = 5 msec, then only 1% of the time is wasted. In this example, the cpu efficiency is .99.

- By using a fixed quantum, q, we assume all processes are equally important. This is frequently not the case.

Priority Scheduling (P)

- each process is assigned a priority, and a runnable process with the highest priority is scheduled next.
- Priorities may be either assigned statically or dynamically.
- E.g., priorities could be reduced every time a process is scheduled to prevent a high priority process from hogging resources.
- In UNIX, you can check the priority of a process, using:
  - `ps -l -u<username>`
  - A process that is CPU-bound is given lower priority (higher PRI number) than a process that is I/O-bound (lower PRI number).
- It is often convenient to group processes with the same priority in a class and use RR scheduling within the class.
- Variant -
  - Multilevel Queue Scheduling - Each queue is assigned a different priority class permanently.

Multilevel feedback queues

- High priority = small q, low priority = large q. Priorities change based on I/O level of process.
- CPU-bound jobs sink to the bottom, while I/O bound jobs stay at the top.
- Policy: To prevent a process that was CPU-bound at first and became interactive later from being punished forever, whenever a carriage return was typed at the terminal, the process belonging to the terminal was moved to the highest priority class thinking that it was about to become interactive.
- Moral: Getting it right in practice is much harder than getting it right in principal.

Miscellanea

- Real-time scheduling - the scheduler makes real promises to the user in terms of deadlines or CPU utilization.
- **Two-level scheduling** - a high-level scheduler decides which processes should be in memory, and a low-level scheduler is used to schedule the processes in memory.