# Processes, Context Switching, and Scheduling

Kevin Webb

Swarthmore College
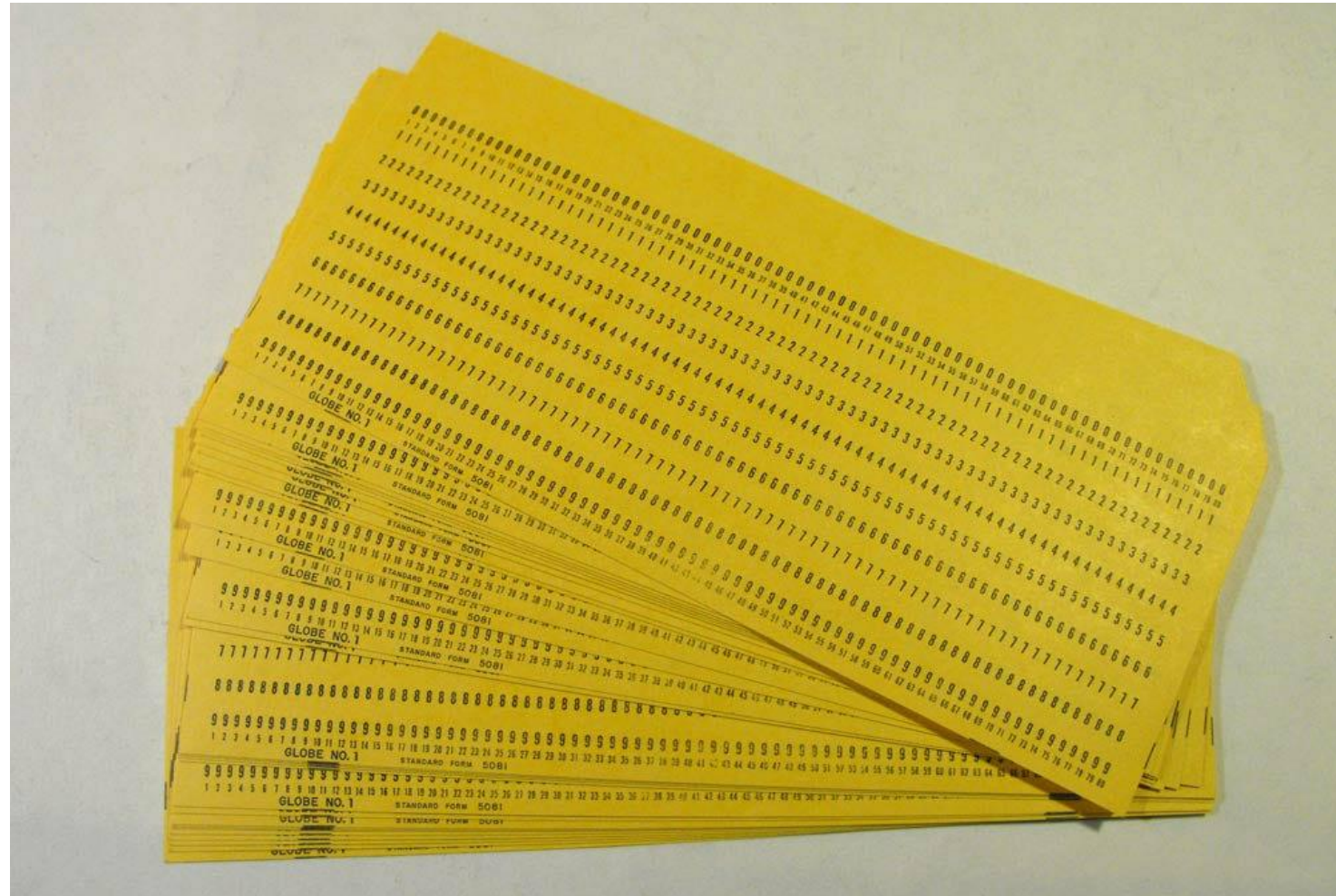
January 30, 2018

# Today's Goals

- What is a process to the OS?

- What are a process's resources and how does it get them?

- In particular: focus on CPU execution
  - Mechanism: context switching and how it works
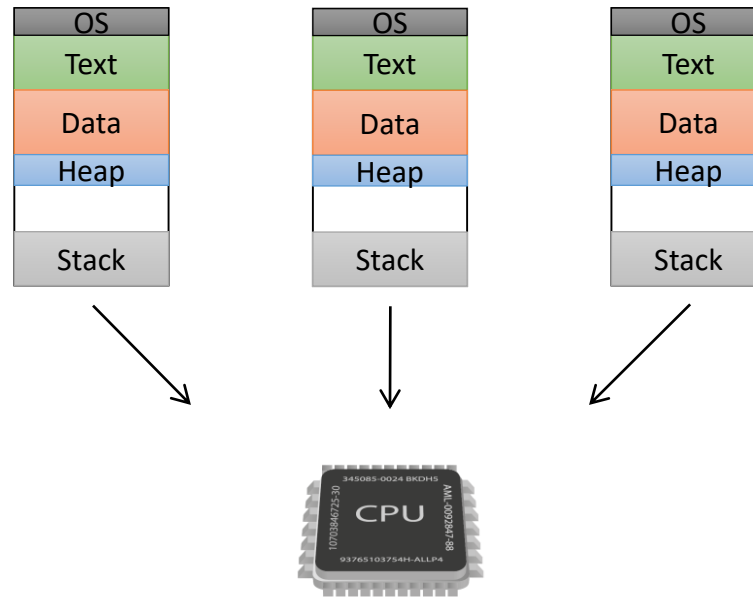  - Policy: CPU scheduling and the types of policies that make sense

# Before Processes (or even OSes)

- Feed in program

- Wait for output

- Feed in next one…

# Goal: Multiprogramming

- Multiprogramming: have multiple programs available to the machine, even if you only have one CPU core that can execute them.

# Why is multiprogramming beneficial if we can still only execute instruction stream at a time?

A. Usability – It's a pain to have only one program.

B. Efficiency – Switching between running programs improves the performance of each program.

C. Efficiency – Switching between running programs improves the performance of the overall system.

D. Cost – Need to buy less hardware if one machine can run everything.

E. Some other reason(s).

# Goal: Multiprogramming

- Multiprogramming: have multiple programs available to the machine, even if you only have one CPU that can execute them

- When I/O issued by process, CPU not needed!
  - Allow another program to run
  - Requires yielding the CPU and sharing memory

- Challenges: what if one running program…
  - Monopolizes CPU, memory?
  - Reads/writes another's memory?
  - Uses I/O device being used by another?

The machine's hardware is NOT exclusive to one running program anymore…

# Solution: Processes

- To the OS, a process is state to be kept track of and protected!
  - For now, let's assume processes have a single thread of execution.


- Need to store (at least):
  - **execution eligibility state** (can it be scheduled right now?)
  - process id – PID
  - parent and child processes
  - **resources** (CPU state, memory, I/O)
  - …

# Process "State"

- "What can this process do right now?"

- Running: process is executing on the CPU

- Ready: process can execute, but we have to give it the CPU

- Waiting / blocked: process is waiting for *something* to happen before it can continue. Does NO GOOD to schedule it.

Examples:
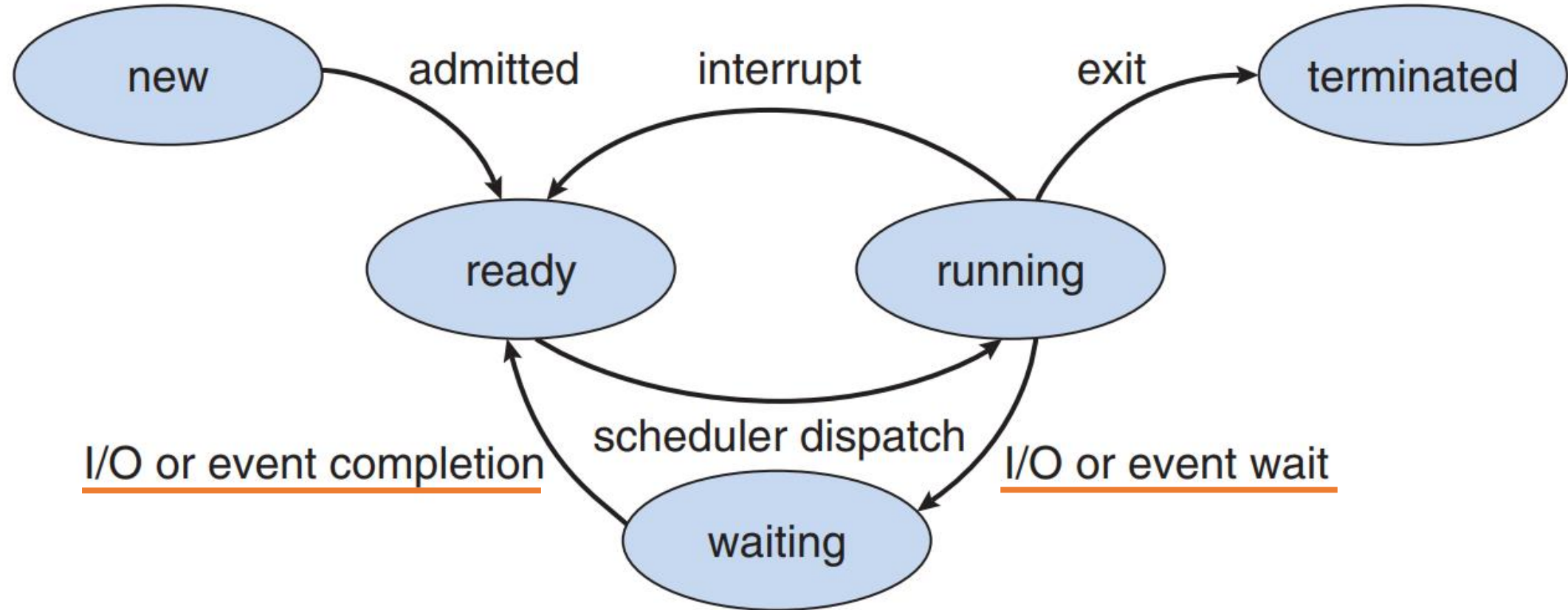Waiting for I/O to complete.
Process needs to wait for exclusive resource (e.g., mutex).
Process asks to be put to sleep for a while…

# It doesn't make sense for a process to go from____.   Why not?

A.   running to waiting/blocked


B.   ready to waiting/blocked


C.   ready to running


D.   running to ready

# State Transitions (& I/O implications)



I/O is important to the scheduler: can't schedule an I/O blocked process.
I/O is important to processes: can't run if they perform I/O!

# Process Resources: Memory

- Abstraction: Virtual Address Space (VAS)

- Give **every process** the illusion of having all of the system's memory. (for convenience!)

- At process startup (fork):
  - Code loaded from disk to *text*
  - Static variables initialized in *data*
  - Stack created in *stack*

- *Heap* allocated on demand (malloc -> syscall)

0x0

| Operating system |
|:---:|
| Text |
| Data |
| Heap |
| |
| Stack |

0xFFFFFFFF
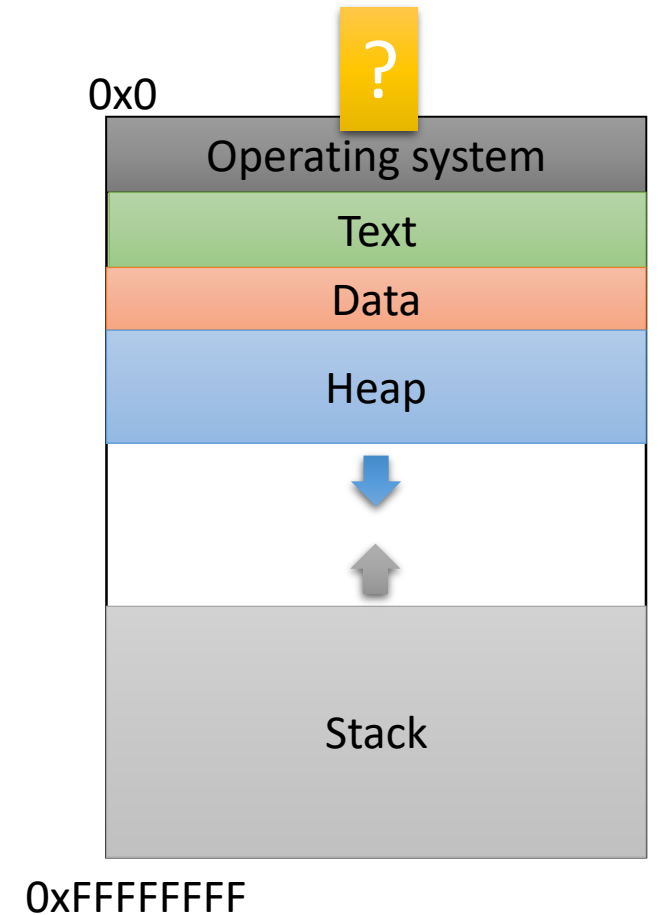
# Process Resources: I/O

- Abstraction: File

- Old Unix adage: "Everything is a file", including:
  - files (duh)
  - sockets (abstraction used for network communication)
  - pipes (send the output of one process to the input of another)
  - most I/O devices (e.g., mouse, printer, graphics card)*

  *Not the only way to access these devices.

# Why treat all of these I/O things as files?

A. It's less error-prone.

B. It provides higher performance.

C. It's simpler to access all of them in the same way.
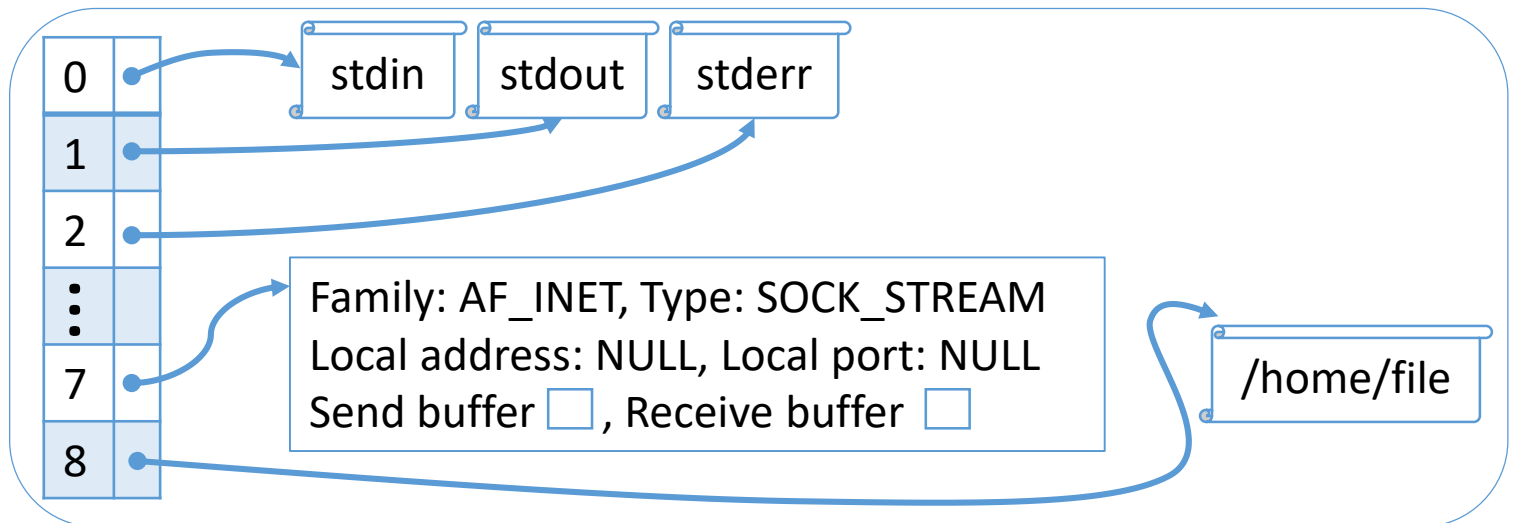
D. More than one of these.

E. Some other reason(s).

# I/O Resource Accounting

- **_For each process_**, OS maintains a *file descriptor table*.
  - Give integer *file descriptor* to process, store details in OS

- By default, processes all get *stdin, stdout, stderr*

*printf* is a `write()` to FD 1 (stdout).

- For anything else, explicitly ask the OS (e.g., `open()`)

| 0 | • | → stdin | stdout | stderr |
| 1 | • |
| 2 | • |
| ⋮ | |
| 7 | • |
| 8 | • |

Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer ☐ , Receive buffer ☐

/home/file

# Lab Note: Shell I/O Redirection Feature

- Shells allow you to redirect one stream to another (or to a file).

- For example:

  - prog 1> output.txt      Store the output (stdout) to a file named output.txt
  - prog 2>&1                    Write stderr to the stdout stream. (Not required for lab)
  - prog < input.txt          Read stdin from input file rather than console.
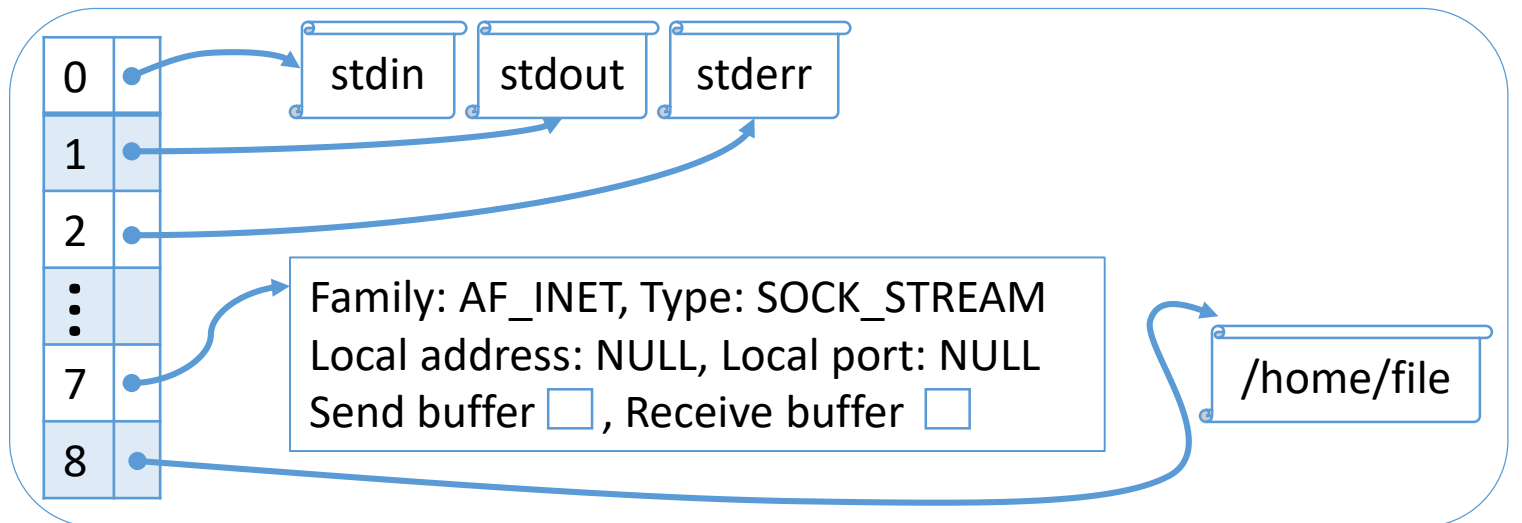
# Process Resource: CPU

- *For each process*, store the process's CPU context:
  - general purpose registers (x86: eax, ecx, etc.)
  - floating point registers
  - program counter register (PC)
  - stack pointer register (SP)
  - memory segmentation registers (x86: CS, DS, SS, ES, FS, GS)
  - any other storage the CPU exposes via ISA…

- Example (Linux x86 thread_struct):
  - https://elixir.free-electrons.com/linux/v4.14.12/source/arch/x86/include/asm/processor.h#L446

# Process Control Block (PCB)

- Lots of things to keep track of *for each process*.
  - Big ones: CPU context, VAS, I/O descriptor table
  - **Lots** of other bookkeeping information

- *Process control block*: per-process structure to centralize everything we know about a process.

- It has many names: PCB, task control block, process table entry, task struct

- Example (Linux task_struct)
  - http://elixir.free-electrons.com/linux/v4.14.12/source/include/linux/sched.h#L519

# Great, each process has a PCB, with all the process's info. We need to choose one of them to execute on the CPU. How?

A. Run one to completion, then choose another one after that.

B. Prioritize according to user's (or system designer's) preference.

C. Give each process the same amount of CPU time.

D. Prioritize by some dynamic performance metric.

E. Some other policy.

# CPU Scheduling: There is no "best" policy...

- Depends on the goals of the system.

- Different for...
  - Super computer
  - Nuclear power plant or medical device
  - Your personal computer

- Often have multiple (conflicting) goals or primary metrics

# Scheduling Metrics

- CPU Utilization – percentage of time CPU is not idle

- Turnaround time – time between process startup and completion
  - Typically concerned with average

- Throughput – how many processes get completed

- Fairness – how well is the CPU distributed among processes

# Which metrics are most important in each scenario? What might a reasonable scheduling policy look like for each? (No clickers, think briefly solo then discuss.)

1. Super computer: large, long running jobs submitted by many users

2. Medical device: sensors for monitoring patient and actuators for doing something to them (e.g., administering medication)

3. General-purpose desktop/laptop: variety of tasks happening for one user (browser, email, music, messaging, etc.)

Metrics: CPU utilization, turnaround time, throughput, fairness, others?

# Observations

- Super computer probably has lots of CPU hungry tasks, not much I/O.

- Task priority probably critical to medical device.

- Humans like interactivity on desktop/laptop, even at the expense of overall runtime.

# Idealized Policies

- First come, first served (FCFS) / FIFO: Run jobs to completion in the order they arrive.
  - metric: simplicity(?), low overhead

- Shortest job first (SJF): Chose the job that needs least CPU time to finish, run it first. Analogy: "ten items or less" checkout line.
  - metric: turnaround time

- Round robin (RR): Each process gets the same amount of CPU time, taking turns fairly.
  - metric: fairness

# More Realistic General-Purpose Policy

- Special class gets special treatment (varies – requires configuration)

- Everything else: *roughly* equal time quantum
  - "Round robin"
  - Give priority boost to processes that frequently perform I/O
  - Why?

- "I/O bound" processes frequently block.
  - If we want them to get equal CPU time, we need to give them the CPU more often.

# Multi-Level Feedback Queue (BSDs, Linux 2.4)

- Multiple ready queues 0, 1, ..., n

- Always select process in lowest-numbered queue

- Run selected process for $2^i$ quantums (for queue i)

- If process doesn't block, place in next higher queue (except last), else back to same queue (or 0)
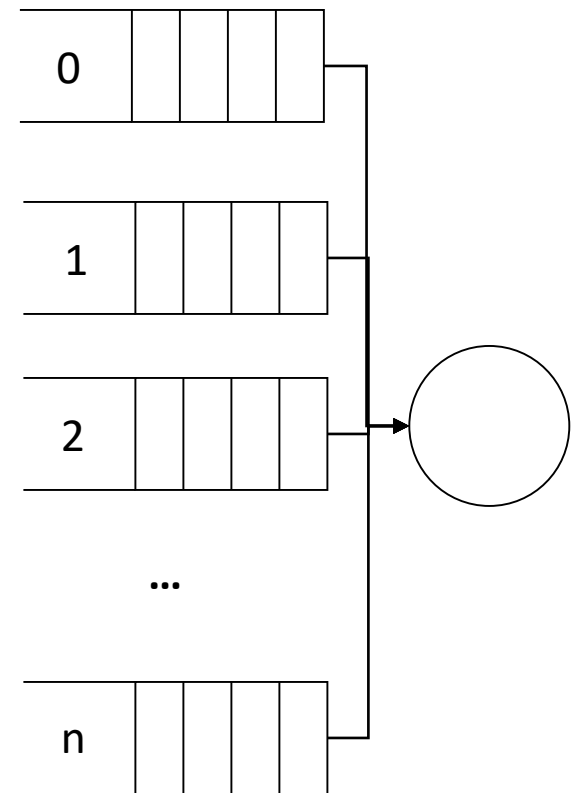
# Linux's "Completely Fair Scheduler"

(default since Oct 2007)

- "real time" process classes – always run first (rare)

- Other processes:
  - Red-black BST of process, organized by CPU time they've received.
  - Pick the ready process that has run for the shortest (normalized) time thus far.
  - Run it, update it's CPU usage time, add to tree.

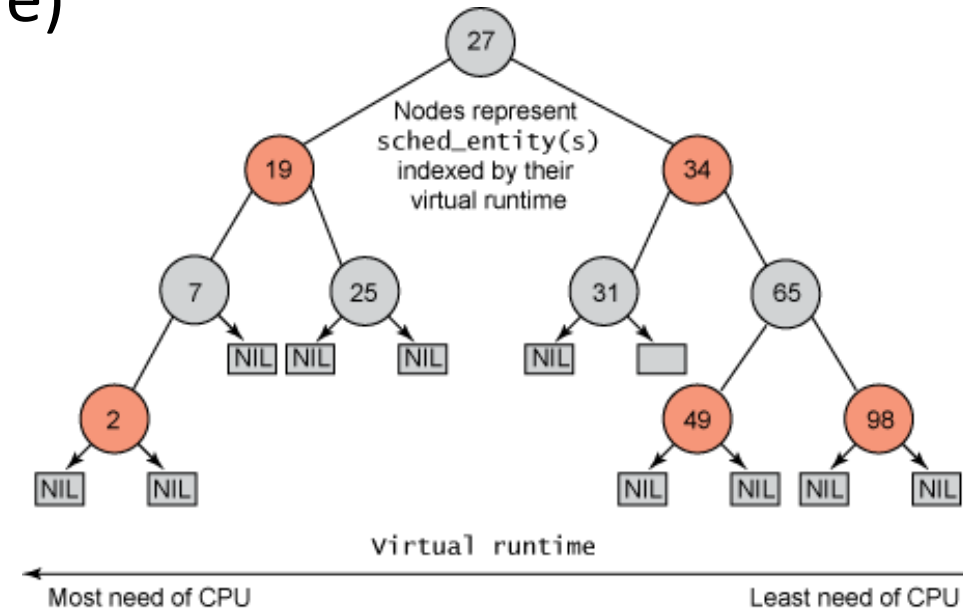- Interactive processes: Usually blocked, low total run time, high priority.



Image source: https://www.ibm.com/developerworks/library/l-completely-fair-scheduler/

# Windows

- "Each thread has a *dynamic priority*. This is the priority the scheduler uses to determine which thread to execute. Initially, a thread's dynamic priority is the same as its base priority. The system can boost and lower the dynamic priority, to ensure that it is responsive and that no threads are starved for processor time."

- Priority is boosted when:
  - Process's window is brought to foreground.
  - Process's window receives input.
  - Process was waiting for I/O, which has now completed.

- Source: https://msdn.microsoft.com/en-us/library/windows/desktop/ms684828(v=vs.85).aspx

# Mechanism: Context Switching

- Regardless of policy, we need to control which process gets CPU!

- Executive summary:
  - First, save CPU context of currently running process to PCB
  - Next, load CPU context of next process to run from PCB
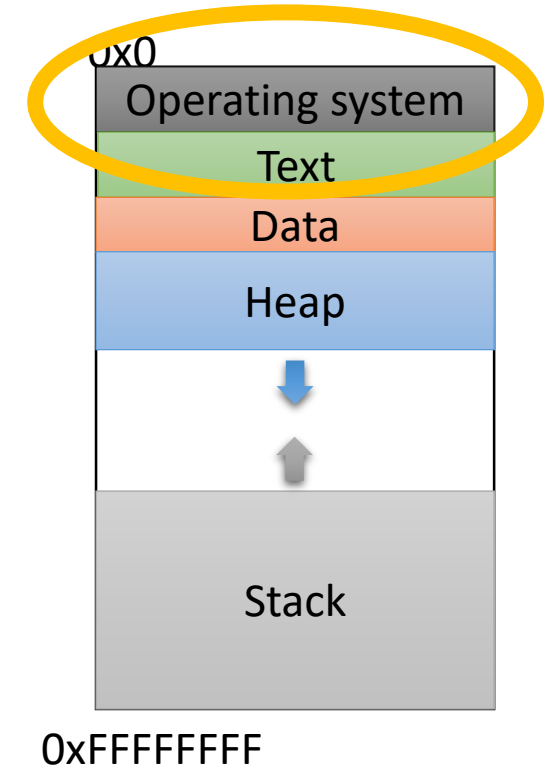
# Context Switching: Performance

- Even though it's fast, context switching is expensive:
    1. time spent is 100% overhead
    2. must invalidate other processes' resources (caches, memory mappings)
    3. kernel must execute – it must be accessible in memory

- Solution to #3:
    - keep kernel mapped in every process VAS
    - protect it to be inaccessible

- Aside: 2018 "meltdown" hardware bug
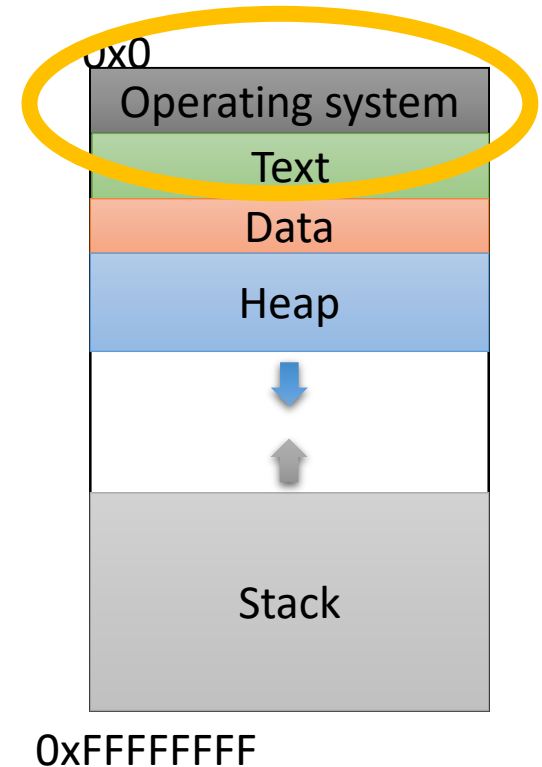
0x0

| Operating system |
| Text |
| Data |
| Heap |
| |
| Stack |

0xFFFFFFFF

# Kernel Execution

- Recall: kernel executes on…
  - process system call
  - process exception
  - hardware interrupt

- Problem: the kernel calls functions too, it needs space to work with.
  - Dynamic memory (Linux): kmalloc / kfree
  - Stack: set aside memory for kernel stack in each process

Every process has *another* stack up here, for use when kernel is executing on its behalf.

0x0

Operating system

Text

Data

Heap

Stack

0xFFFFFFFF

## Userspace Process *P*

```
main() {                    Text
    int i = 10;
    yield();

    i = 20;
    yield();
}
```

```
                            Stack
i:   ?
```

Scenario: Process *P* causes itself to be context switched by calling a `yield()` system call.

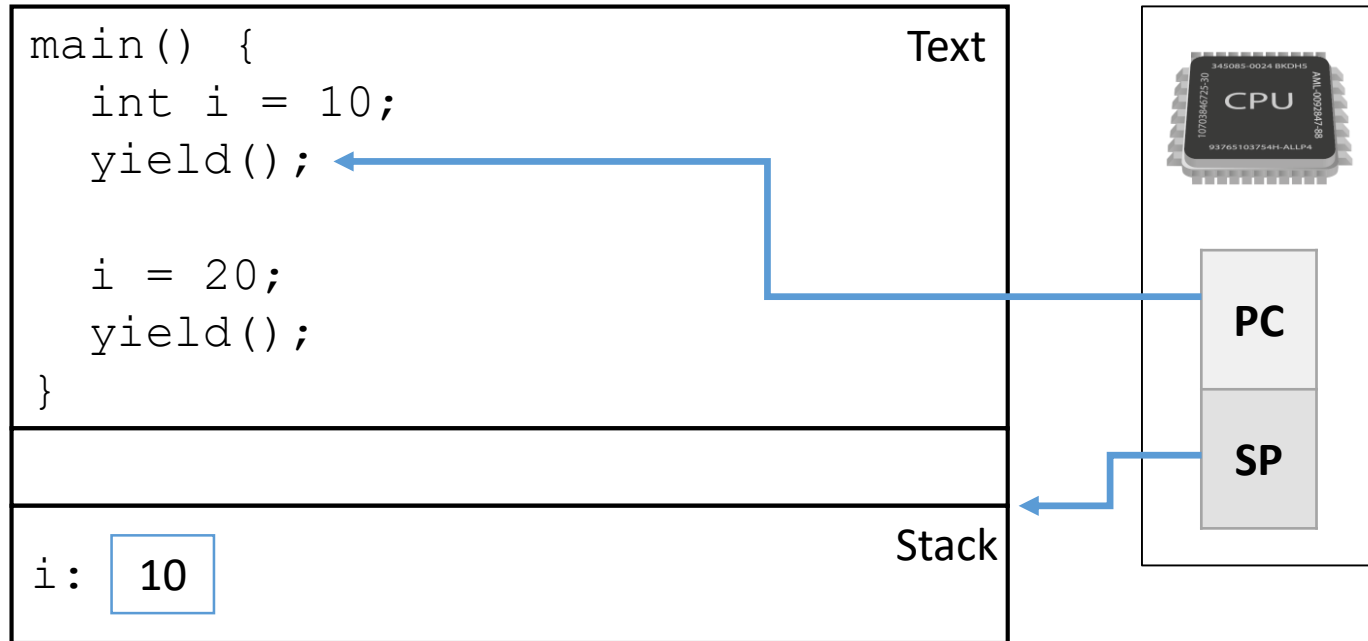`yield()` – just give up the CPU / voluntary context switch.

## Userspace Process *P*

```
main() {
    int i = 10;
    yield();

    i = 20;
    yield();
}
```

Text

Stack

i: [ ? ]

**PC**

**SP**
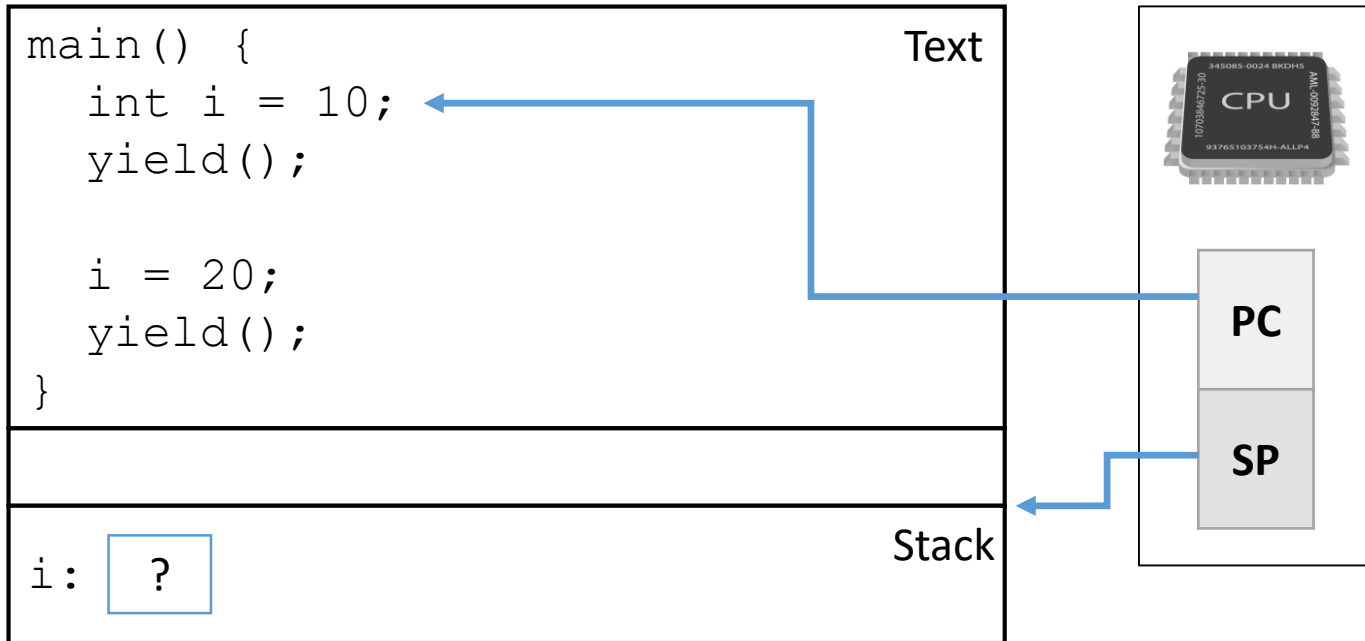
Process *P* starts up and is given the CPU.

PC: next instruction to execute
SP: top of stack

## Userspace Process *P*

```
main() {
    int i = 10;
    yield();

    i = 20;
    yield();
}
```

Text

Stack

i:  10

CPU

PC

SP

Process *P* executes the first instruction.

Userspace Process *P*
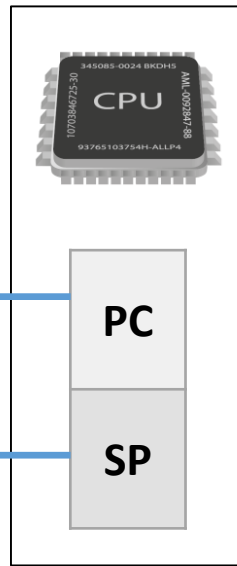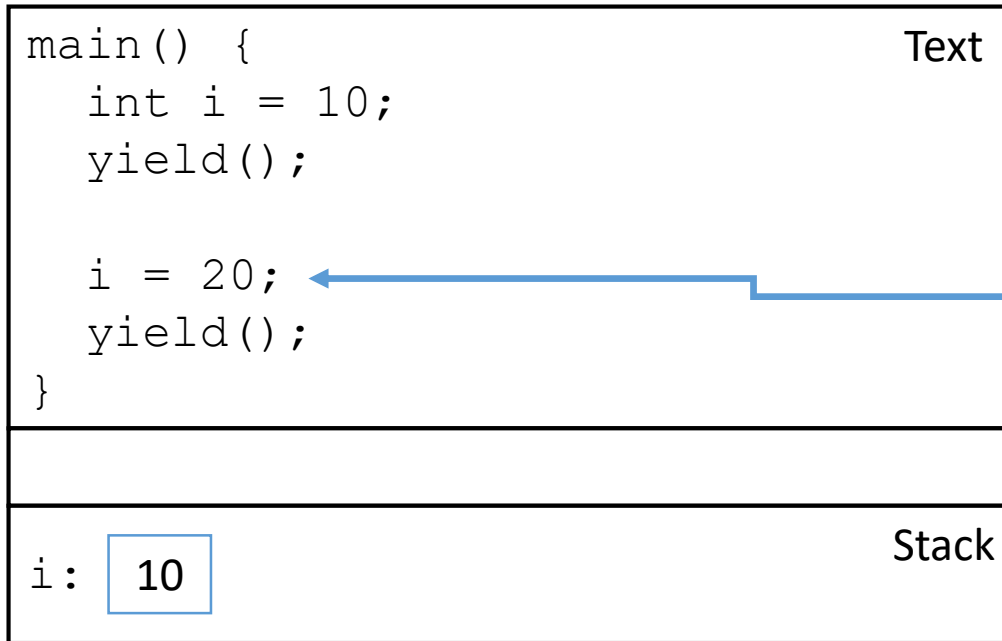
```
main() {                    Text
    int i = 10;
    yield();

    i = 20;
    yield();
}
```
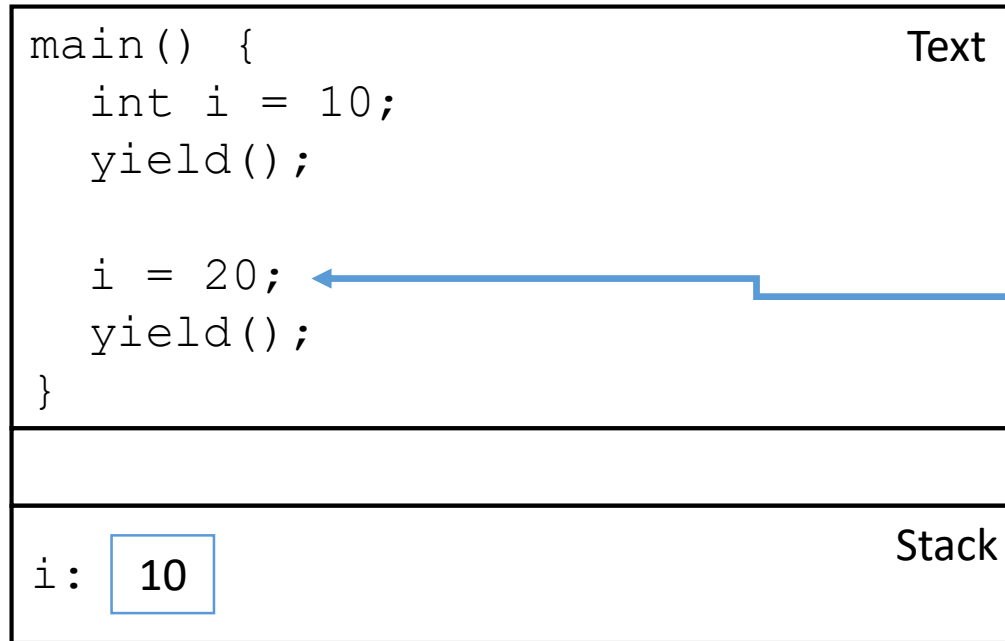
Stack

i:  10

PC

SP

Process *P* makes a system call: yield.

Transition to kernel mode:
- **change CPU "ring" setting**

OS

## Userspace Process P

```
main() {                          Text
    int i = 10;
    yield();

    i = 20;
    yield();
}
```

```
                                  Stack
i:  10
```

**CPU**

PC

SP
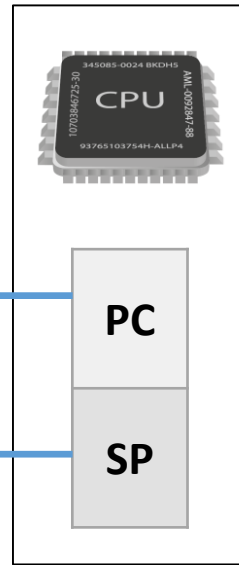
Process P makes a system call: yield.
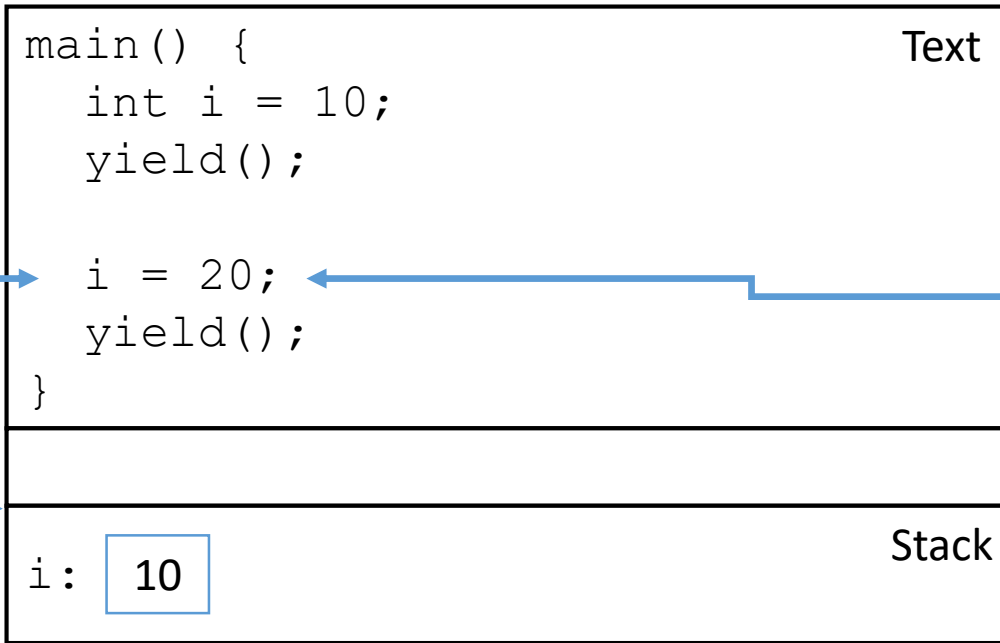
Transition to kernel mode:
- change CPU "ring" setting
- **find system call entry point**

```
yield() {
    pid_t old, new;
    old = current_pid();
    new = schedule();
    context_switch(old, new);
}
```

OS

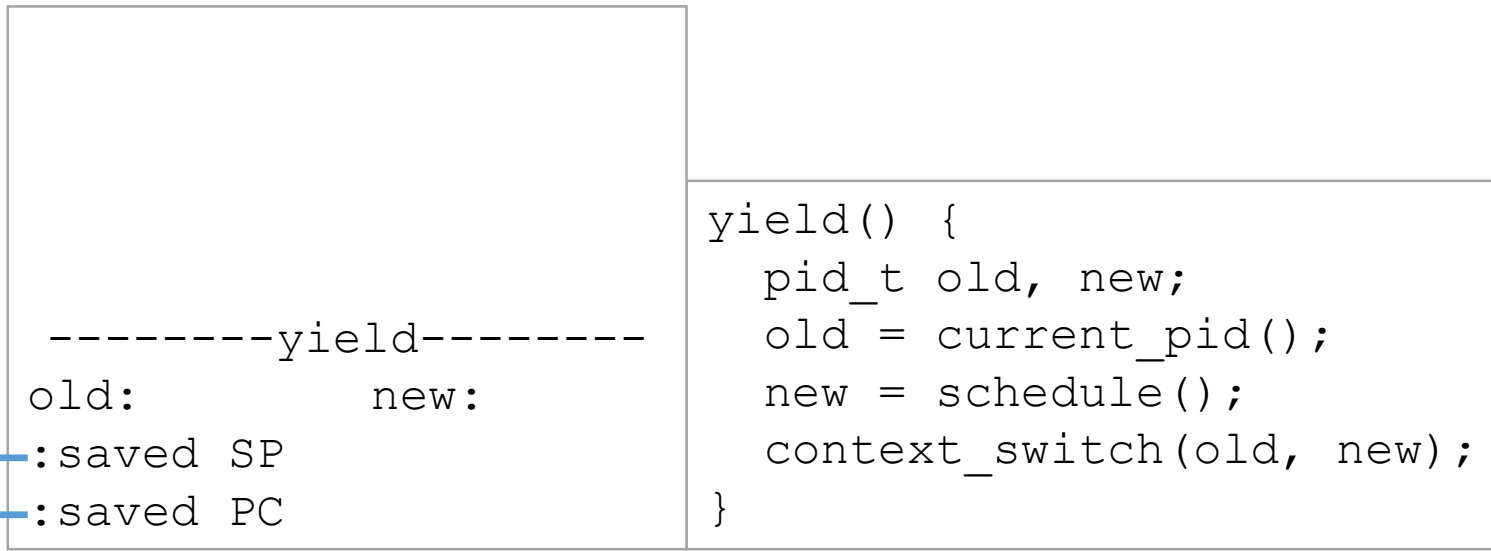This is a caricature of a system call for learning purposes!

Userspace Process *P*
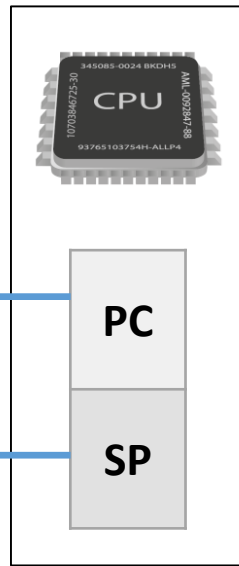
```
main() {                               Text
  int i = 10;
  yield();

  i = 20;
  yield();
}
```

```
                                      Stack
i:   10
```

CPU

**PC**

**SP**

Process *P* makes a system call: yield.

Transition to kernel mode:
- change CPU "ring" setting
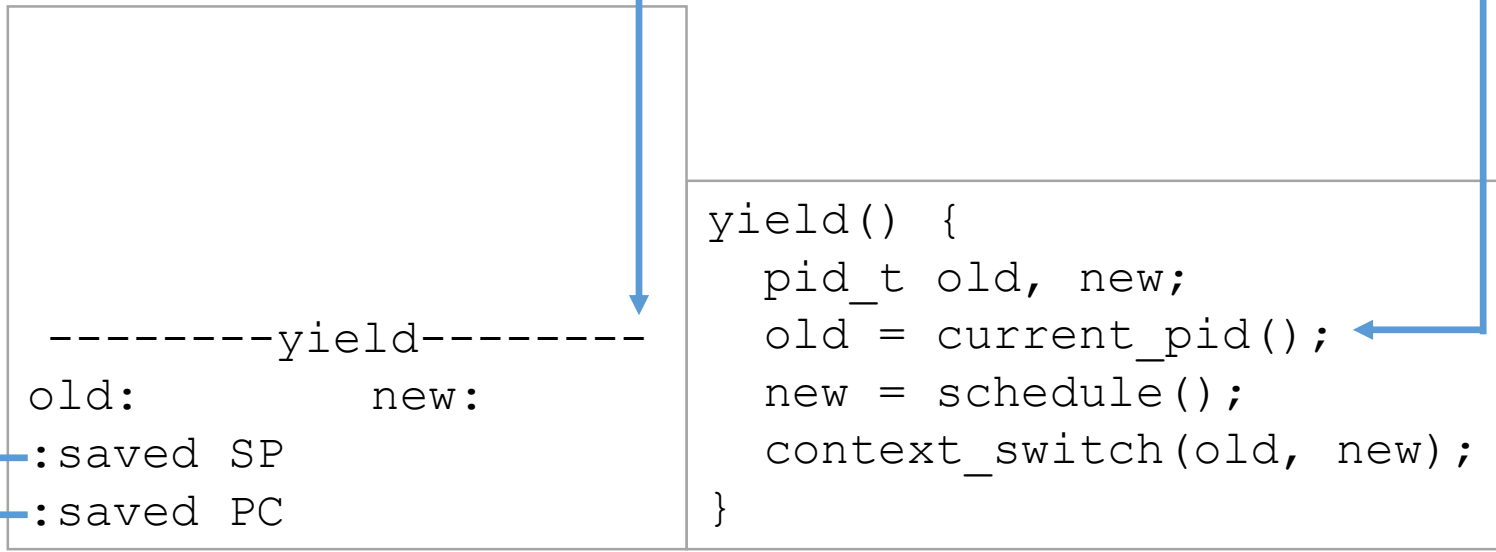- find system call entry point
- **set up kernel stack**

```
        --------yield--------
old:          new:
:saved SP
:saved PC
```

P's kernel stack

```
yield() {
  pid_t old, new;
  old = current_pid();
  new = schedule();
  context_switch(old, new);
}
```

OS

Userspace Process *P*

```
main() {                                    Text
   int i = 10;
   yield();

   i = 20;
   yield();
}
```

Stack

`i:` 10
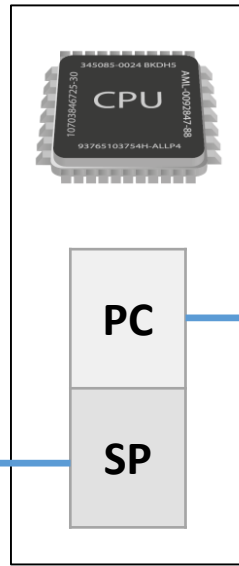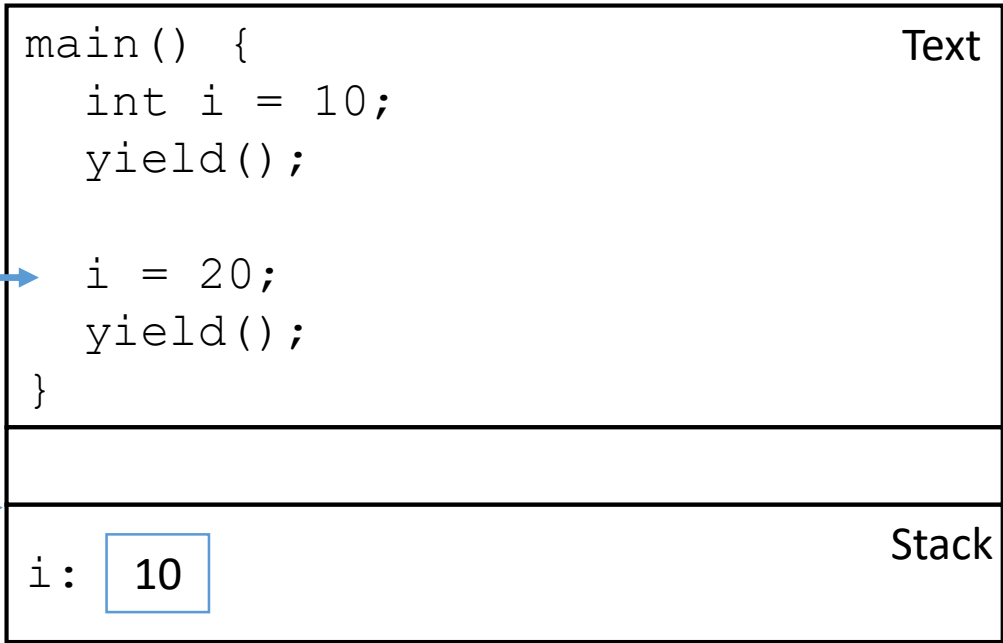
Process *P* makes a system call: yield.

Jump to yield().

**PC**

**SP**

P's kernel stack

```
     --------yield--------
 old:           new:
 :saved SP
 :saved PC
```

OS

```
yield() {
   pid_t old, new;
   old = current_pid();
   new = schedule();
   context_switch(old, new);
}
```

Userspace Process *P*

```
main() {                                Text
  int i = 10;
  yield();

  i = 20;
  yield();
}
```

```
                                        Stack
i:  [ 10 ]
```

CPU

PC

SP
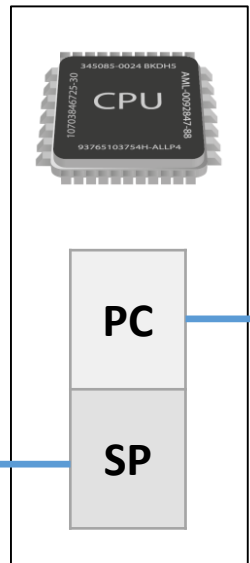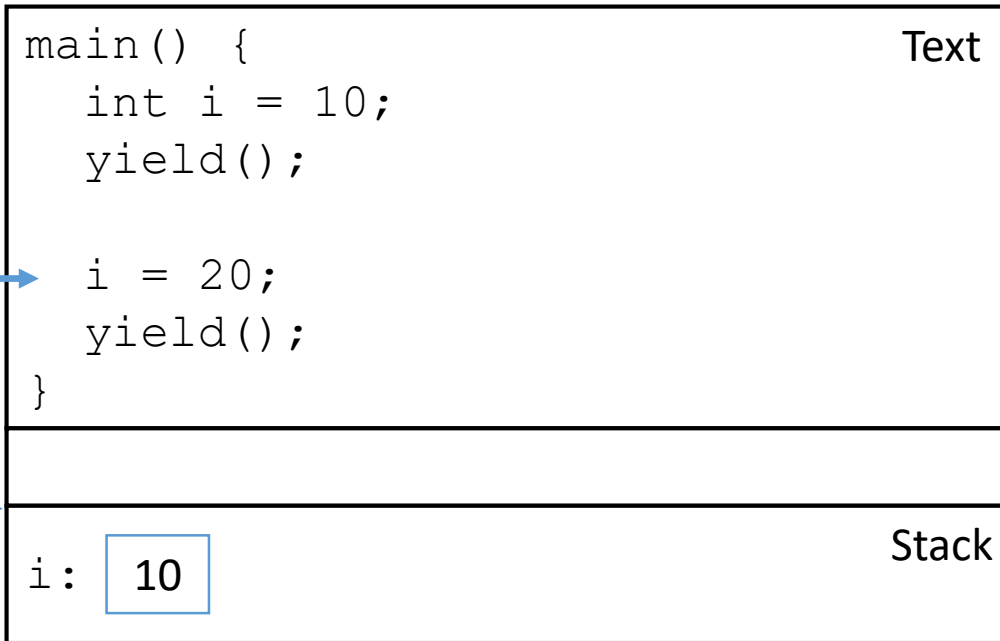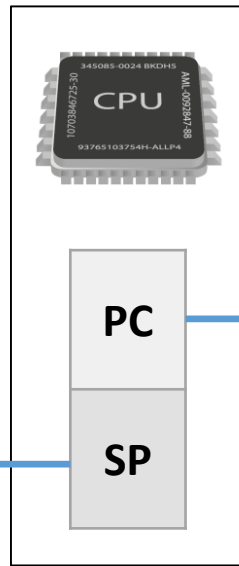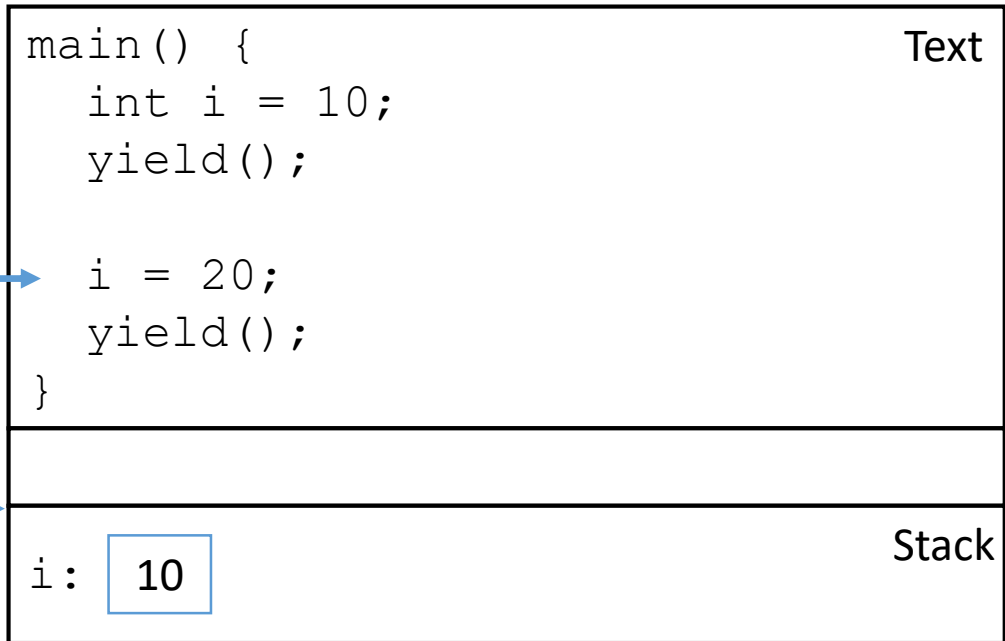
Process *P* makes a system call: yield.

We'll skip over setting old and new.
old: current process
new: process to switch to

```
yield() {
  pid_t old, new;
  old = current_pid();
  new = schedule();
  context_switch(old, new);
}
```

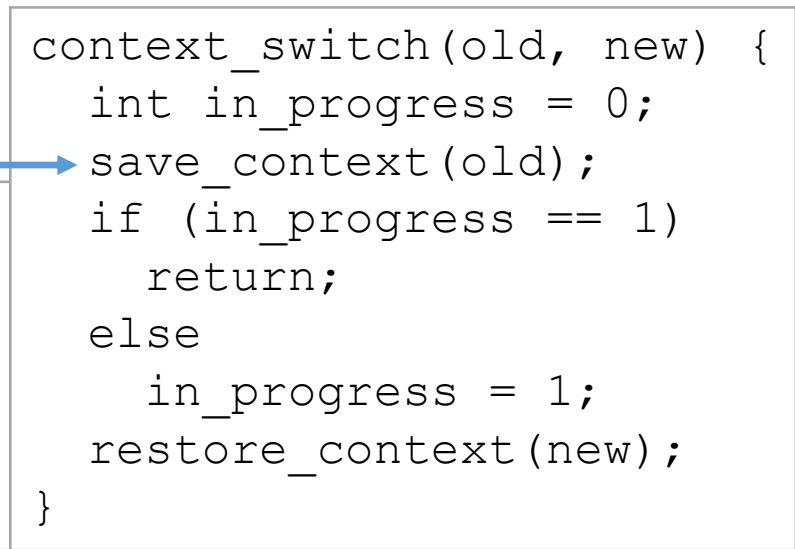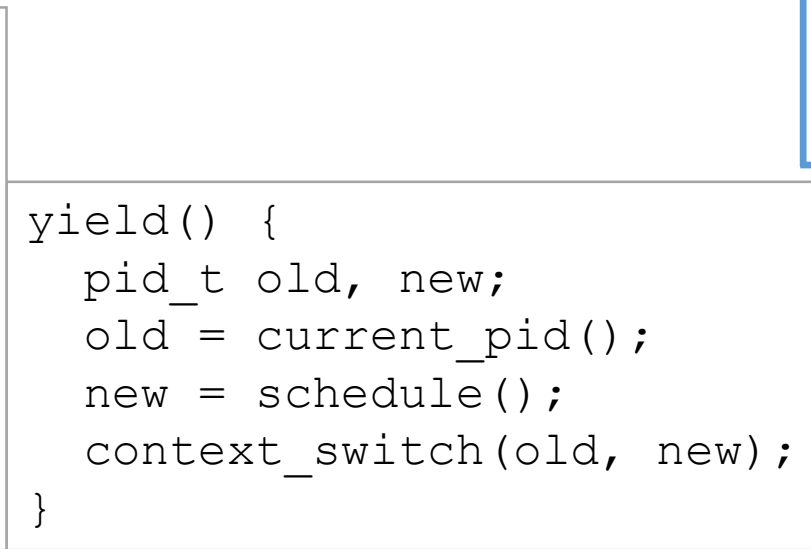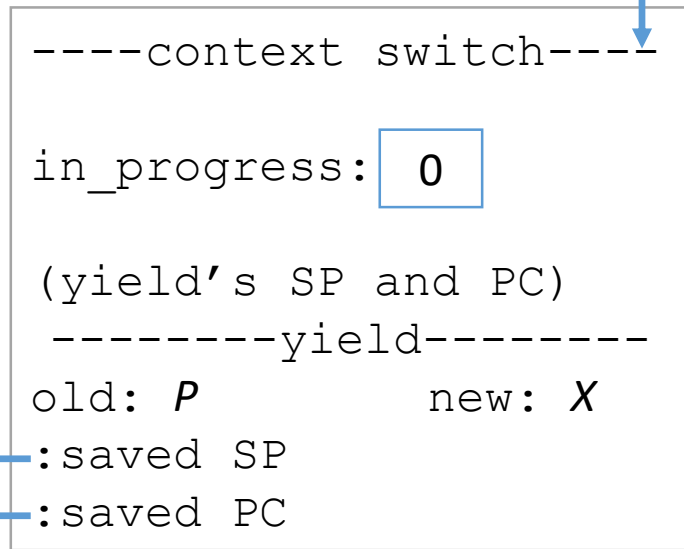```
    --------yield--------
old: P          new: X
:saved SP
:saved PC
```

P's kernel stack

OS

Userspace Process *P*

```
main() {                              Text
  int i = 10;
  yield();


  i = 20;
  yield();

}

                                     Stack

i:  [ 10 ]
```
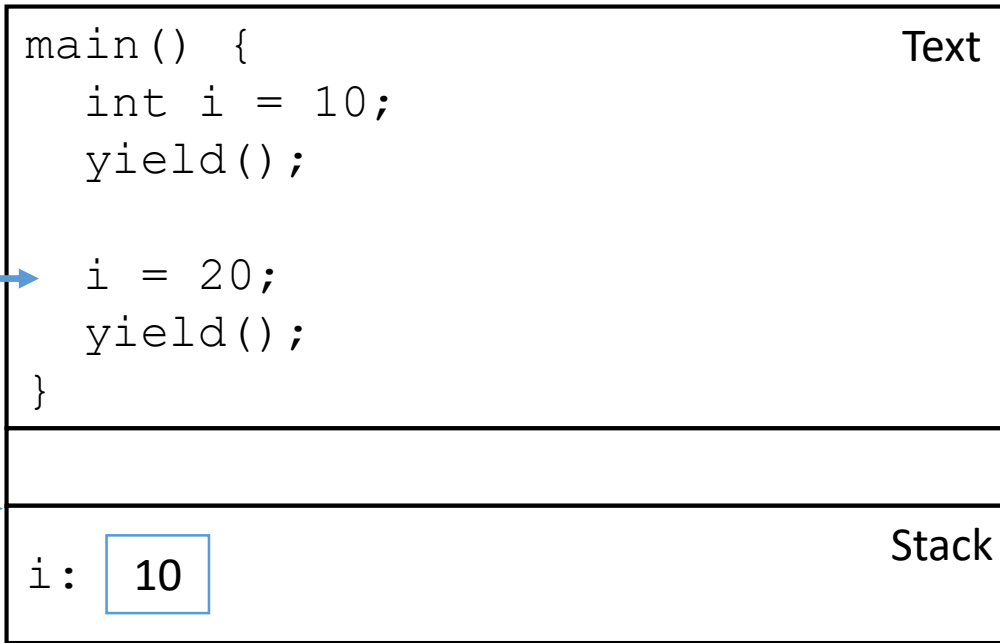
CPU

PC
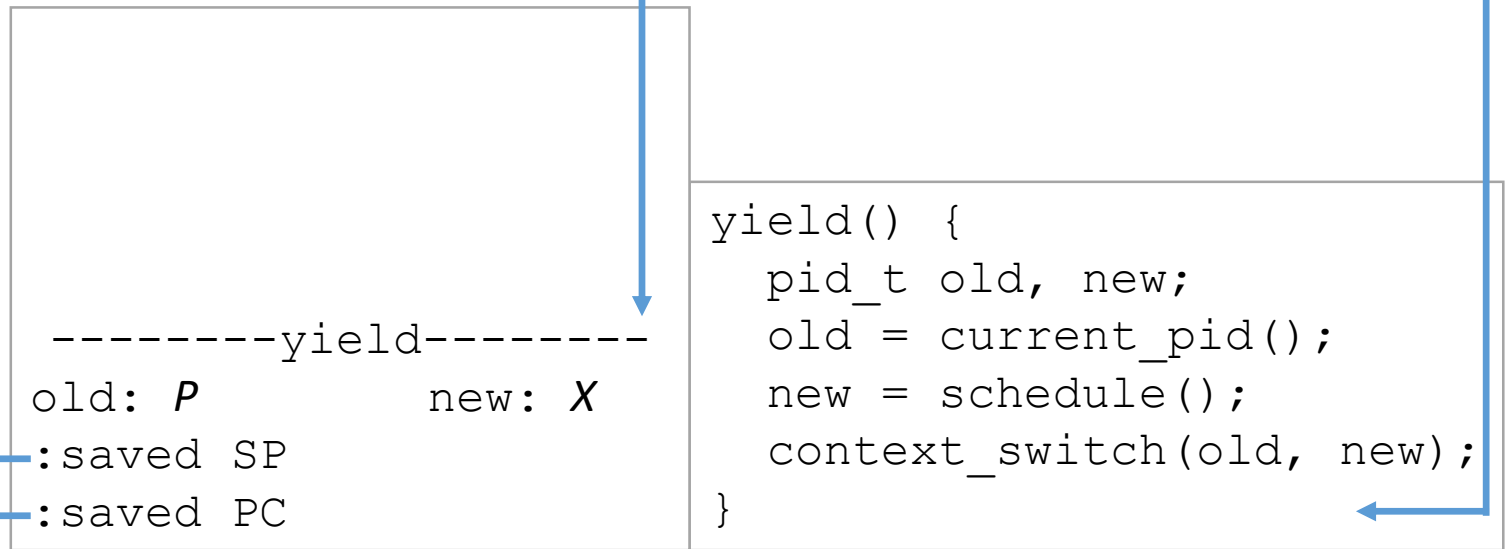
SP
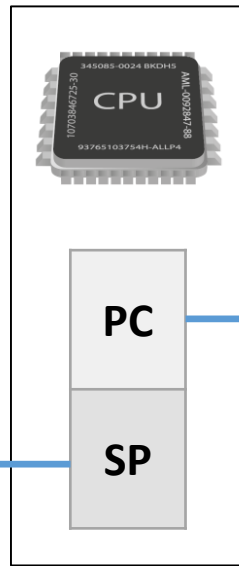
Call context_switch().
   - initialize in_progress to 0

```
----context switch----

in_progress:  [ 0 ]

 (yield's SP and PC)
 --------yield--------
old: P        new: X
:saved SP
:saved PC
```

P's kernel stack

```
yield() {
  pid_t old, new;
  old = current_pid();
  new = schedule();
  context_switch(old, new);
}
```

```
context_switch(old, new) {
  int in_progress = 0;
  save_context(old);
  if (in_progress == 1)
    return;
  else
    in_progress = 1;
  restore_context(new);
}
```

OS

Userspace Process *P*

```
main() {                          Text
  int i = 10;
  yield();

  i = 20;
  yield();
}

                                  Stack
i:  10
```
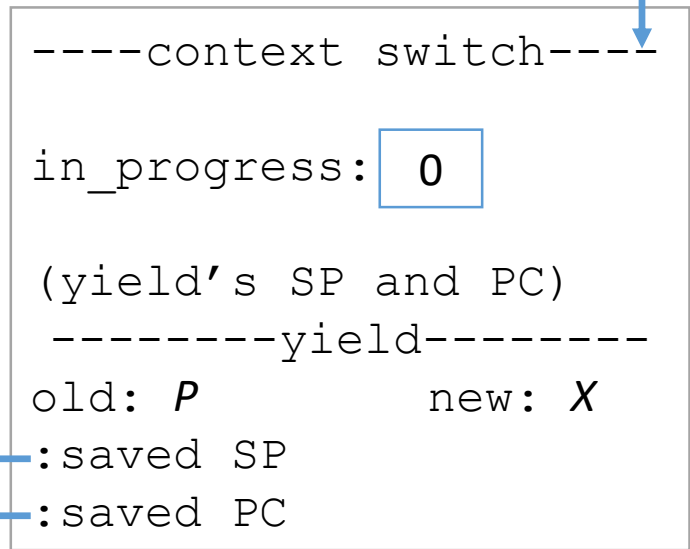
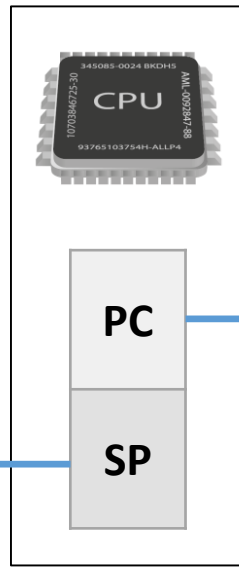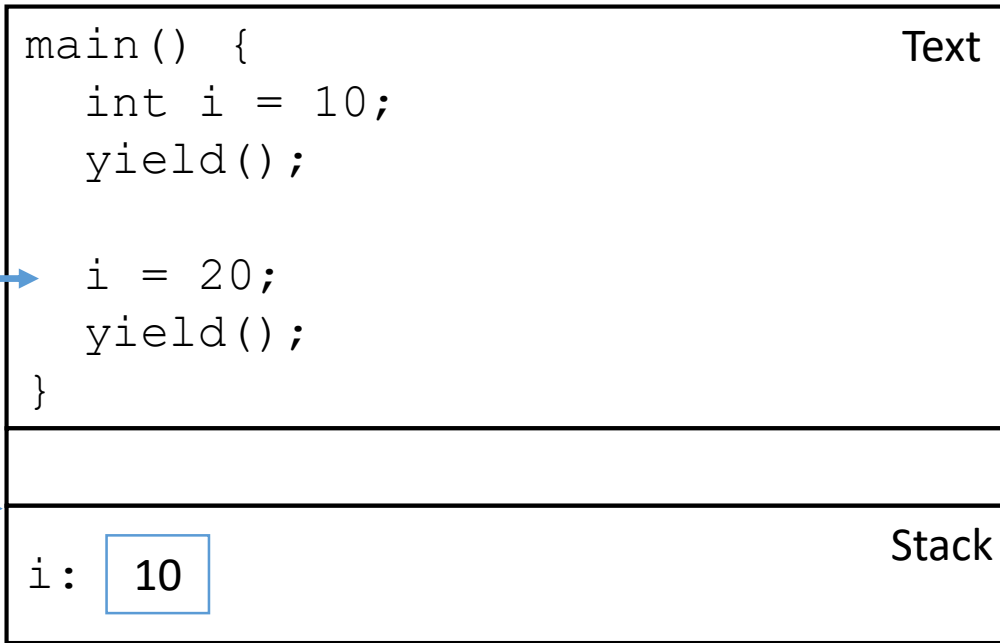Save current process *P*'s context to *P*'s PCB.

CPU

PC

SP

At the time of the save, PC is pointing to the end of save_context(), a return instruction!
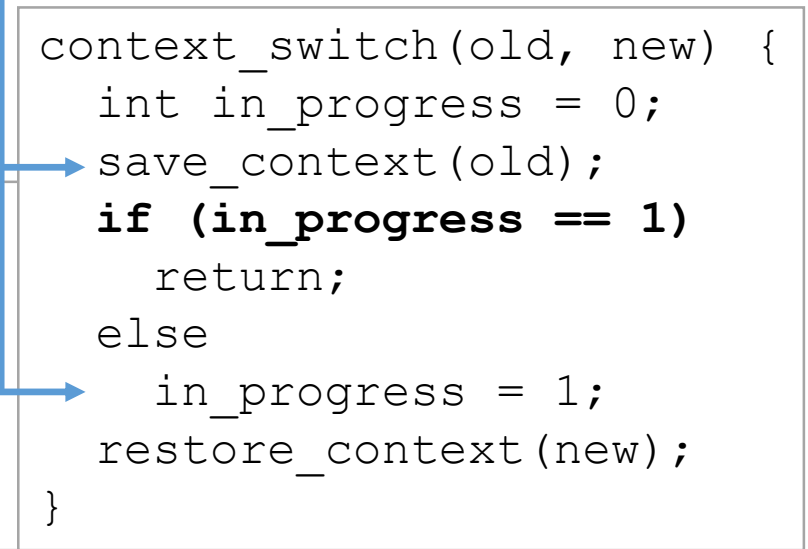
```
----context switch----

in_progress:  0

(yield's SP and PC)
--------yield--------
old: P        new: X
:saved SP
:saved PC
```

P's kernel stack

PCB (*P*): SP, PC

```
yield() {
  pid_t old, new;
  old = current_pid();
  new = schedule();
  context_switch(old, new);
}
```

OS

```
context_switch(old, new) {
  int in_progress = 0;
  save_context(old);
  if (in_progress == 1)
    return;
  else
    in_progress = 1;
  restore_context(new);
}
```

# Userspace Process *P*

```
main() {                                    Text
  int i = 10;
  yield();

  i = 20;
  yield();
}
```

```
                                            Stack
i:    10
```

CPU

PC

SP

```
----context switch----

in_progress:    0

(yield's SP and PC)
--------yield--------
old: P          new: X
:saved SP
:saved PC
```

P's kernel stack

PCB (*P*): SP, PC

```
yield() {
  pid_t old, new;
  old = current_pid();
  new = schedule();
  context_switch(old, new);
}
```

OS

```
context_switch(old, new) {
  int in_progress = 0;
  save_context(old);
  if (in_progress == 1)
    return;
  else
    in_progress = 1;
  restore_context(new);
}
```

## Userspace Process *P*

```
main() {                          Text
  int i = 10;
  yield();

  i = 20;
  yield();
}
```
```
                                  Stack

i:   10
```

CPU

**PC**

**SP**

Set in_progress to 1!

Key insight: we just changed a variable on the stack AFTER saving the context!

```
----context switch----

in_progress:   1

(yield's SP and PC)
--------yield--------
old: P         new: X
:saved SP
:saved PC
```

P's kernel stack

PCB (*P*): SP, PC

```
yield() {
  pid_t old, new;
  old = current_pid();
  new = schedule();
  context_switch(old, new);
}
```

```
context_switch(old, new) {
  int in_progress = 0;
  save_context(old);
  if (in_progress == 1)
    return;
  else
    in_progress = 1;
  restore_context(new);
}
```
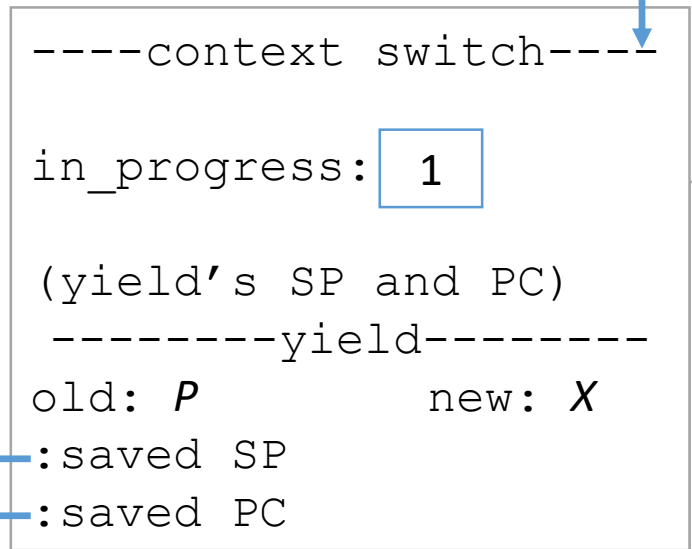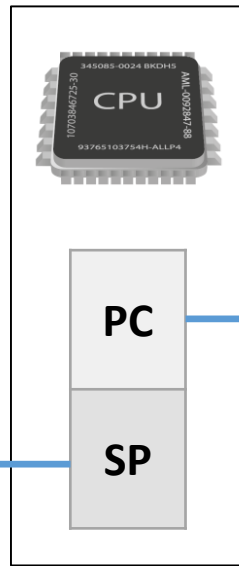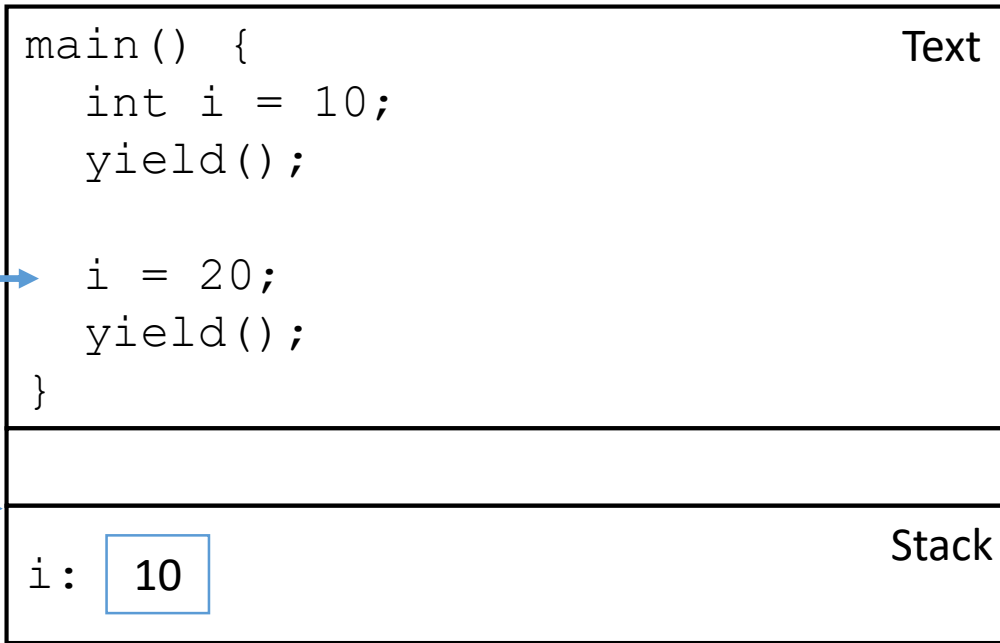
OS

# Userspace Process *P*

```
main() {                                    Text
  int i = 10;
  yield();


  i = 20;
  yield();

}
```

```
                                           Stack

i:    10
```

CPU

PC
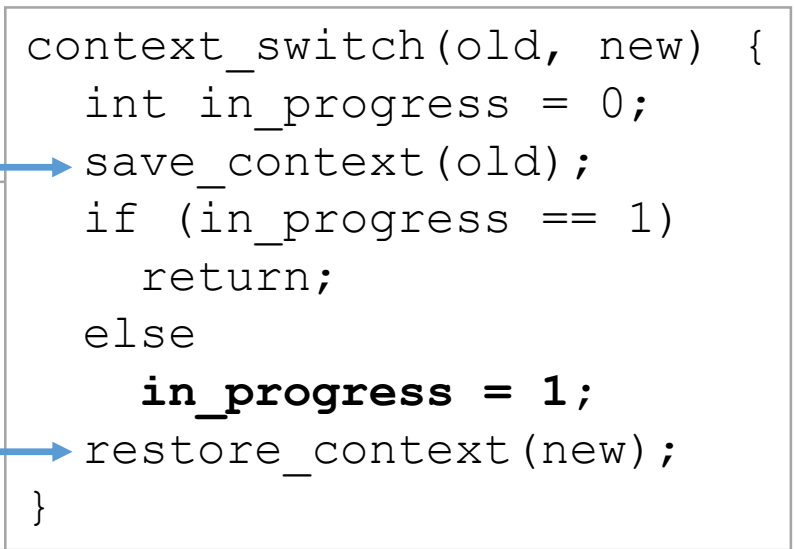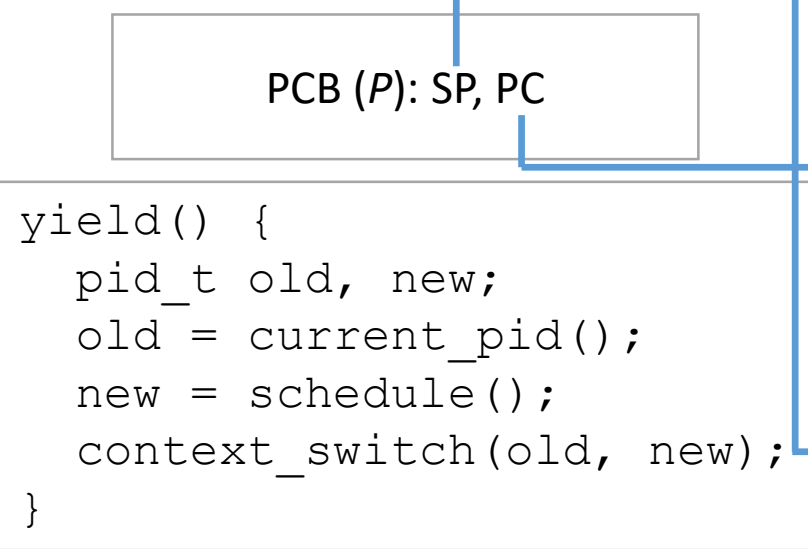
SP

Restore another process's context.

---

```
 ----context switch----

in_progress:   1

 (yield's SP and PC)
  --------yield--------
old: P          new: X

:saved SP

:saved PC
```
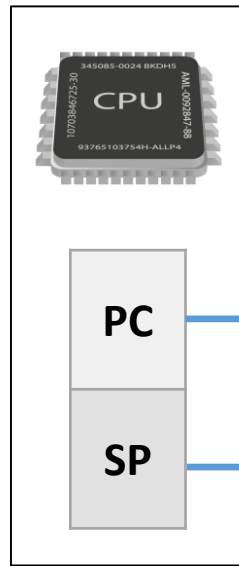
P's kernel stack

```
         PCB (P): SP, PC

yield() {
  pid_t old, new;
  old = current_pid();
  new = schedule();
  context_switch(old, new);
}
```

OS

```
context_switch(old, new) {
  int in_progress = 0;
  save_context(old);
  if (in_progress == 1)
    return;
  else
    in_progress = 1;
  restore_context(new);
}
```

## Userspace Process *P*

```
main() {                              Text
  int i = 10;
  yield();

  i = 20;
  yield();
}
```

```
                                      Stack
i:  10
```
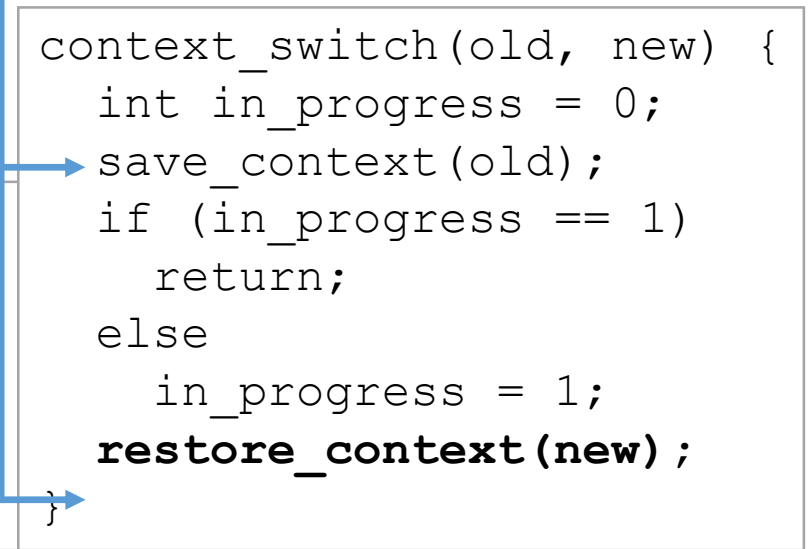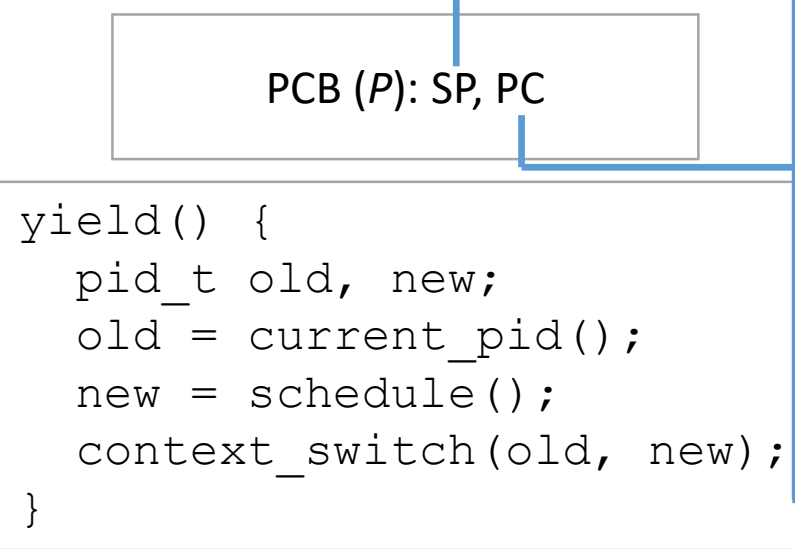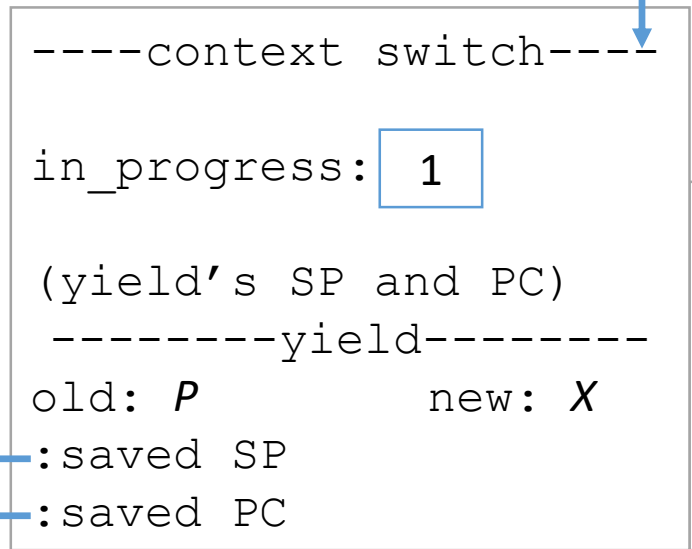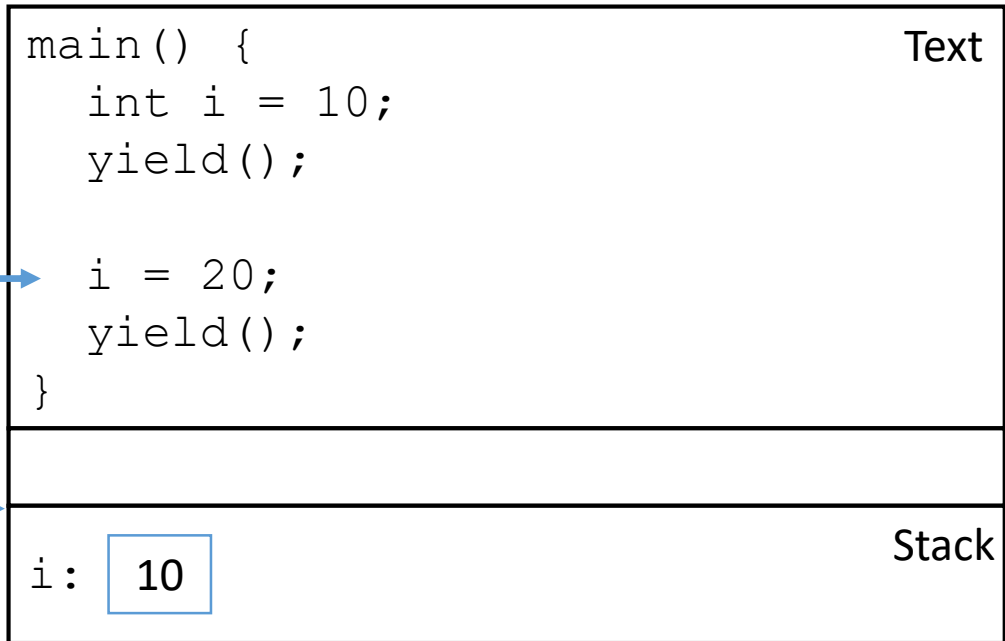
Run other processes for a while...

CPU

PC →

SP →

Process X

---

P's kernel stack

```
 ----context switch----

in_progress:  1

 (yield's SP and PC)
  --------yield--------
old: P          new: X
:saved SP
:saved PC
```

PCB (*P*): SP, PC

```
yield() {
  pid_t old, new;
  old = current_pid();
  new = schedule();
  context_switch(old, new);
}
```

```
context_switch(old, new) {
  int in_progress = 0;
  save_context(old);
  if (in_progress == 1)
    return;
  else
    in_progress = 1;
  restore_context(new);
}
```

OS

# Userspace Process *P*

```
main() {                              Text
  int i = 10;
  yield();

  i = 20;
  yield();
}
```

```
                                      Stack
i:  10
```

**CPU**

PC
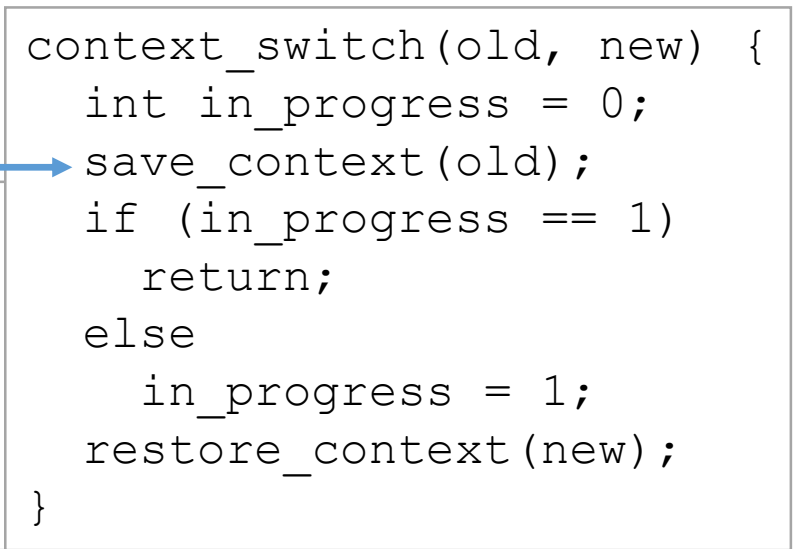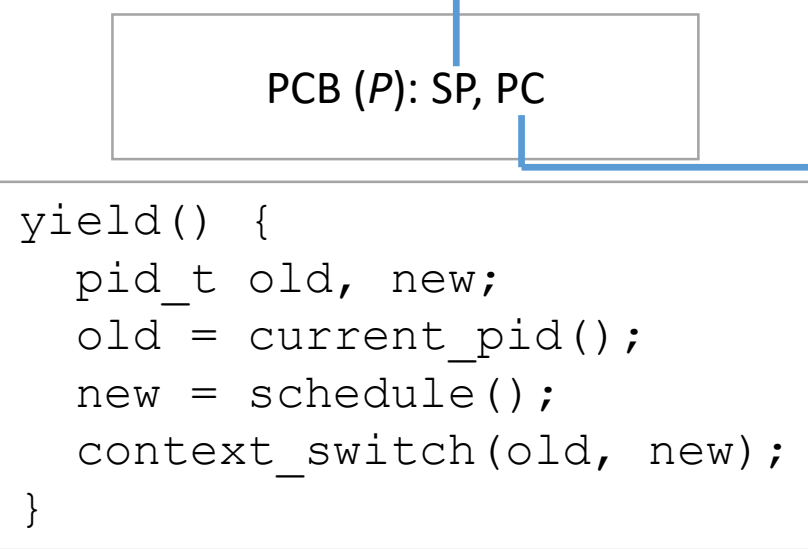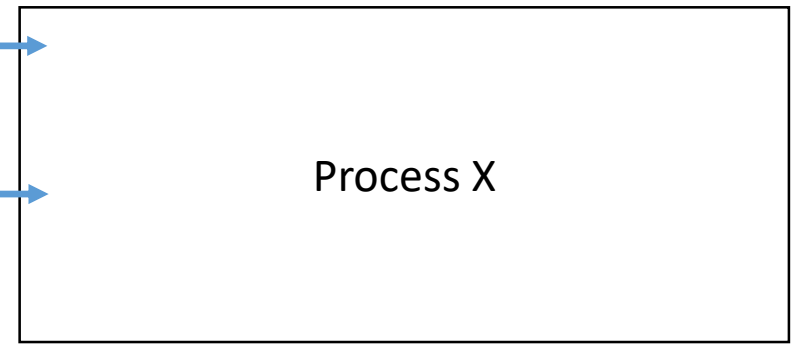
SP
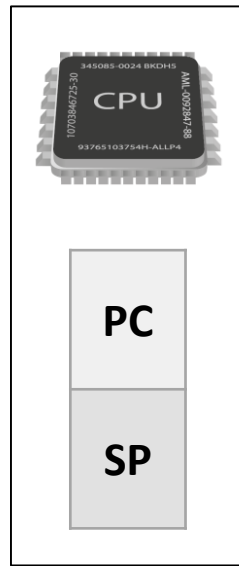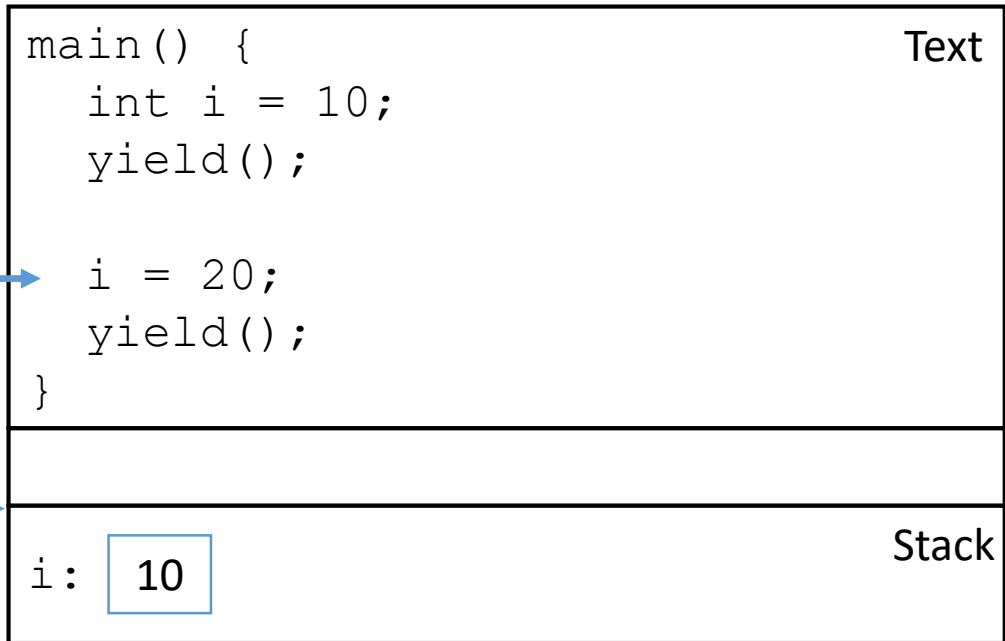
Eventually, we context switch back to process *P* when some other process does a restore_context().

```
----context switch----

in_progress:  1

 (yield's SP and PC)
 --------yield--------
old: P          new: X
:saved SP
:saved PC
```

P's kernel stack

```
   PCB (P): SP, PC

yield() {
  pid_t old, new;
  old = current_pid();
  new = schedule();
  context_switch(old, new);
}
```

OS

```
context_switch(old, new) {
  int in_progress = 0;
  save_context(old);
  if (in_progress == 1)
    return;
  else
    in_progress = 1;
  restore_context(new);
}
```

## Userspace Process P

```
main() {                                    Text
   int i = 10;
   yield();


   i = 20;
   yield();
}
```
```
                                            Stack

i:  [ 10 ]
```

CPU

**PC**
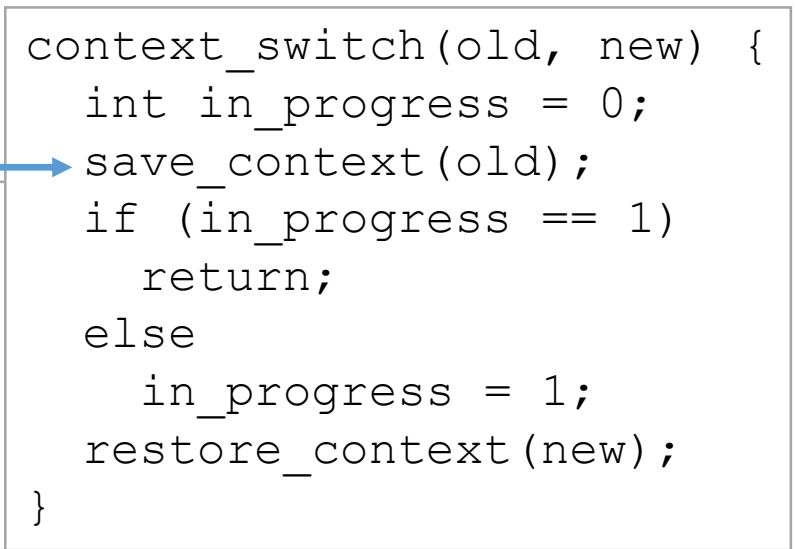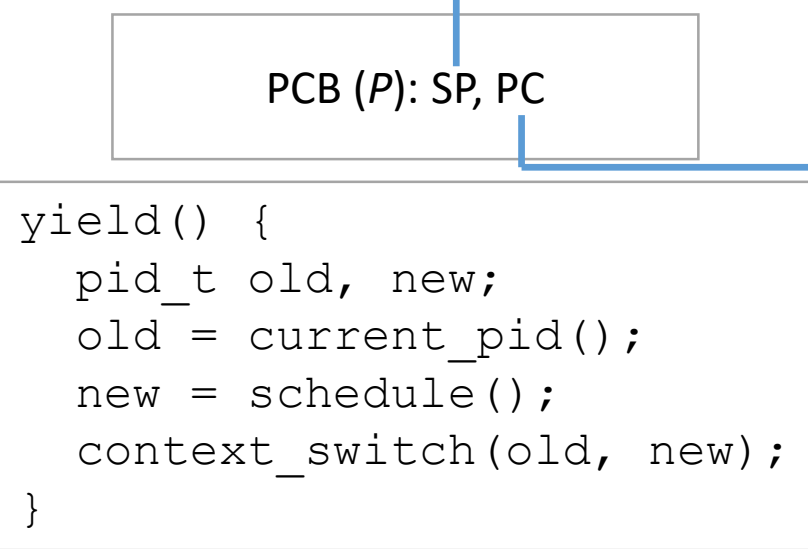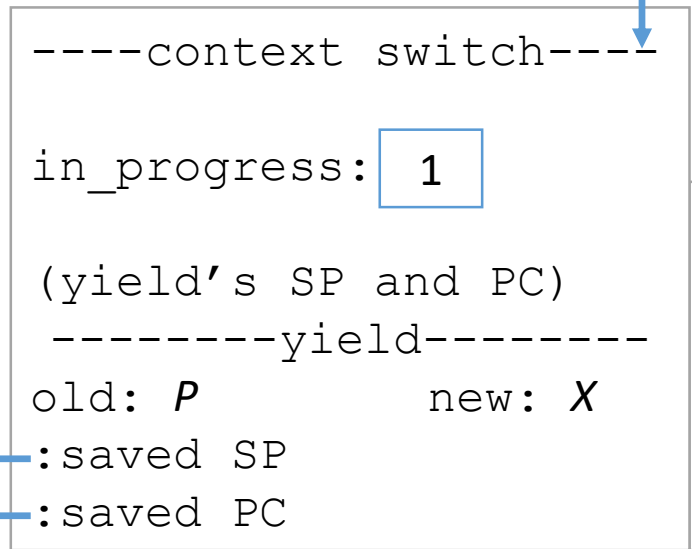
**SP**
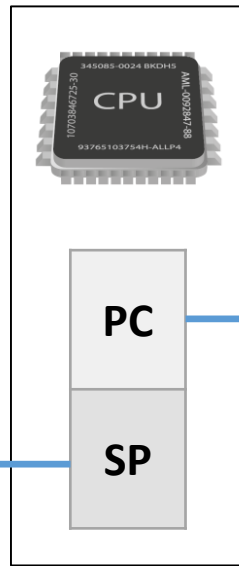
Resume Process P by loading context from PCB.

Recall: PC was pointing at return from save_context, so return back to context_switch().

```
----context switch----

in_progress:  [ 1 ]

 (yield's SP and PC)
 --------yield--------
old: P          new: X
:saved SP
:saved PC
```

PCB (P): SP, PC

```
yield() {
   pid_t old, new;
   old = current_pid();
   new = schedule();
   context_switch(old, new);
}
```

```
context_switch(old, new) {
   int in_progress = 0;
   save_context(old);
   if (in_progress == 1)
      return;
   else
      in_progress = 1;
   restore_context(new);
}
```

P's kernel stack                                    OS

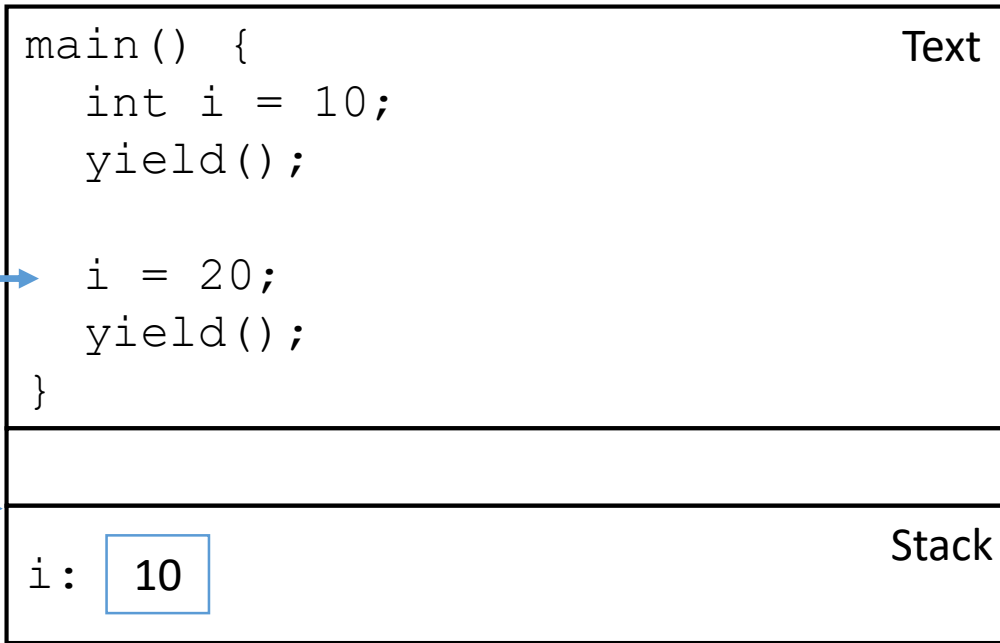## Userspace Process *P*

```
main() {                              Text
    int i = 10;
    yield();


    i = 20;
    yield();

}


                                      Stack

i:   10
```
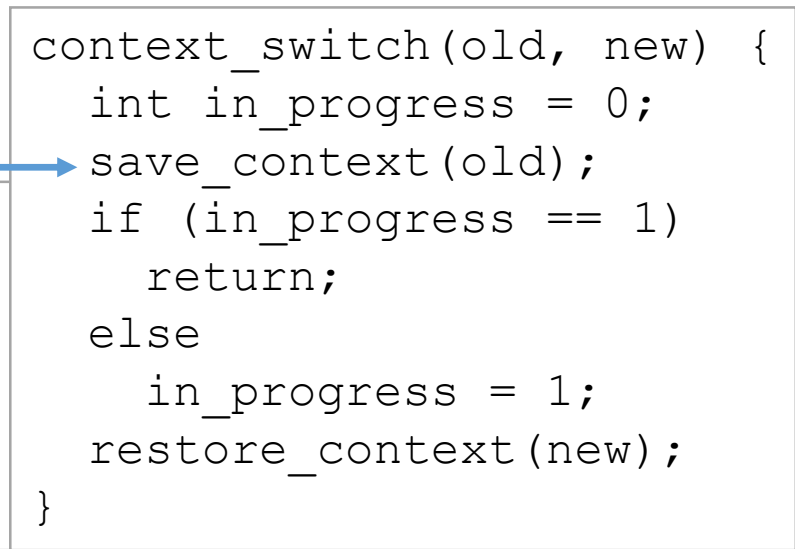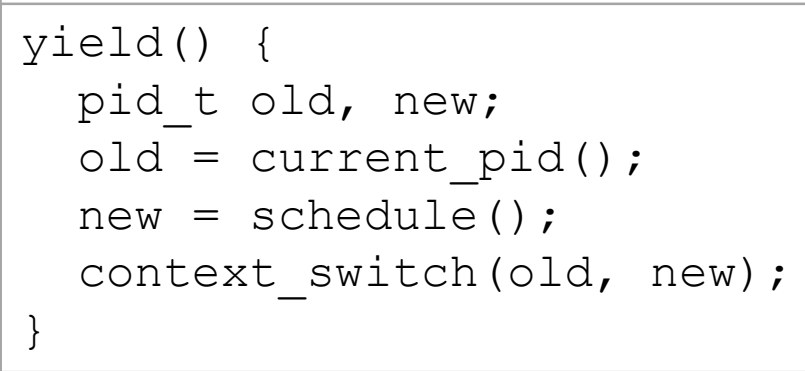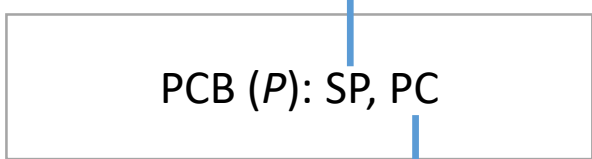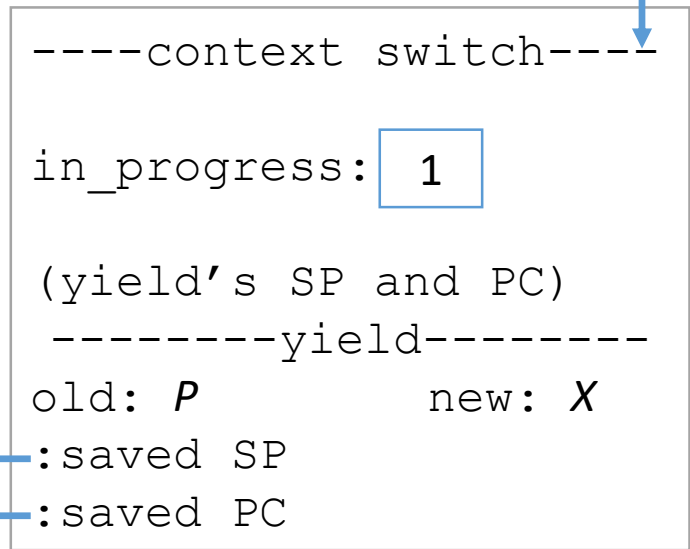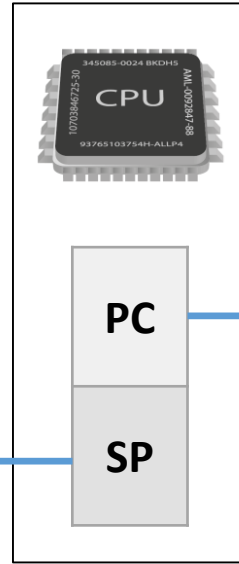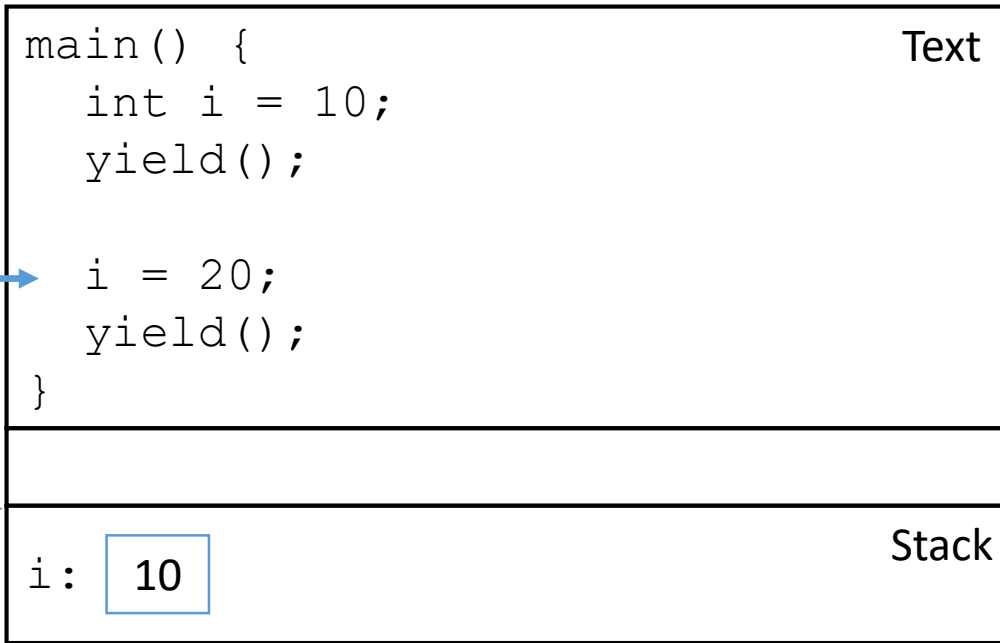
Check in_progress.  It's 1 now!

**PC**

**SP**

```
----context switch----

in_progress:   1

 (yield's SP and PC)
 --------yield--------
old: P        new: X
:saved SP
:saved PC
```

P's kernel stack

PCB (*P*): SP, PC
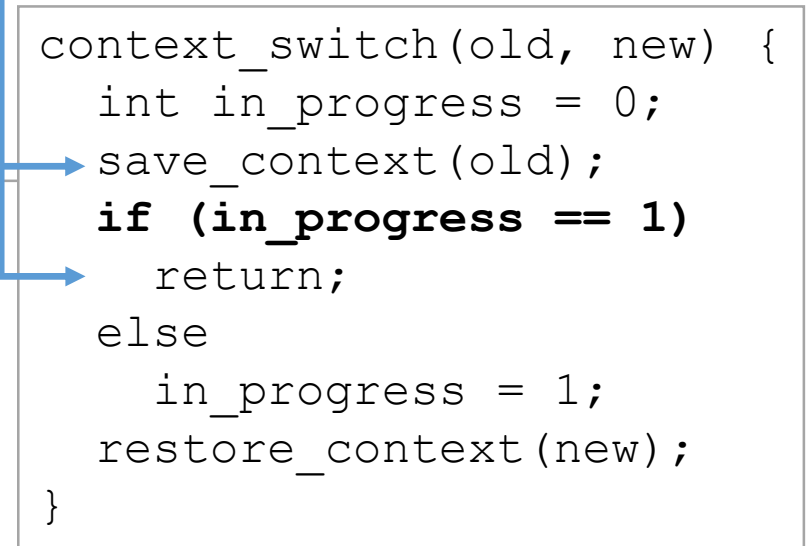
```
yield() {
    pid_t old, new;
    old = current_pid();
    new = schedule();
    context_switch(old, new);
}
```

OS

```
context_switch(old, new) {
    int in_progress = 0;
    save_context(old);
    if (in_progress == 1)
        return;
    else
        in_progress = 1;
    restore_context(new);
}
```

Userspace Process *P*

```
main() {                                    Text
    int i = 10;
    yield();

    i = 20;
    yield();

}
```

                                            Stack

```
i:   10
```

CPU

PC
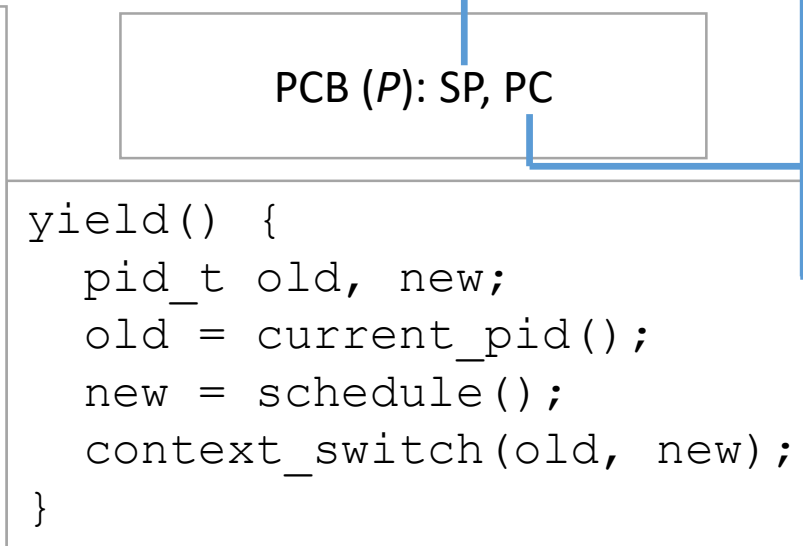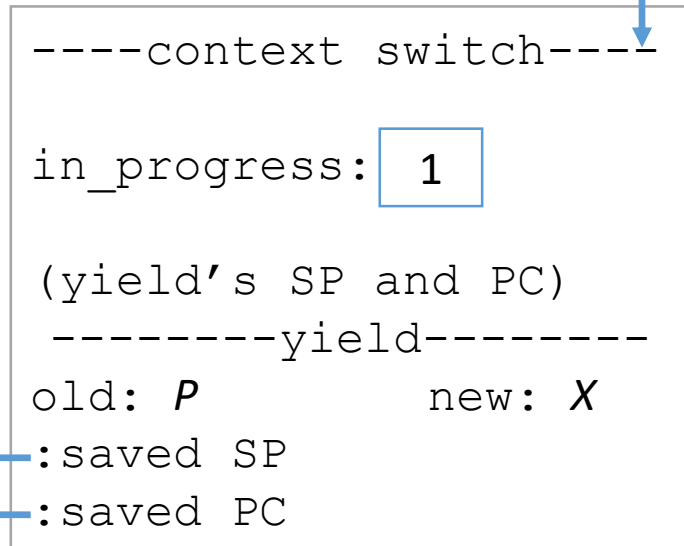
SP
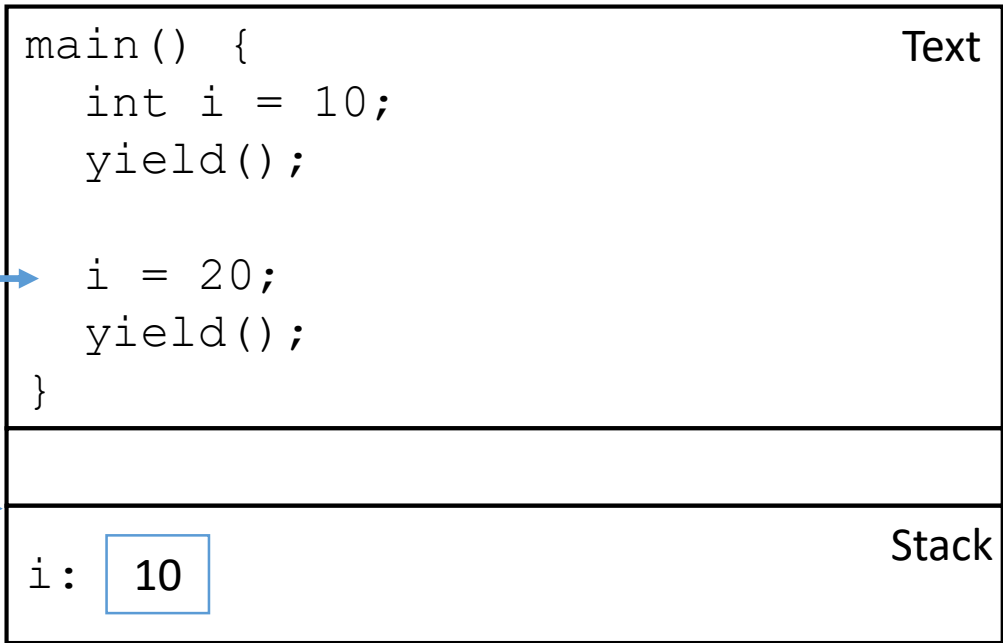
Return back to yield().
Don't need PCB context anymore, so
I'm hiding it (declutter).
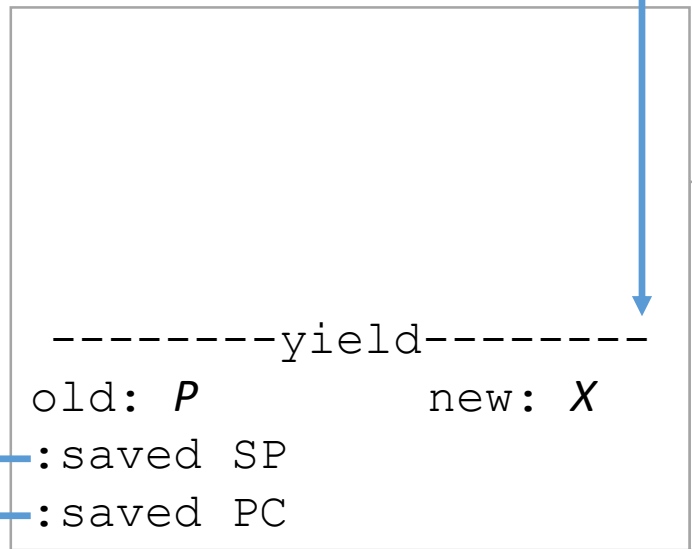
```
    ---------yield---------
old: P          new: X
:saved SP
:saved PC
```
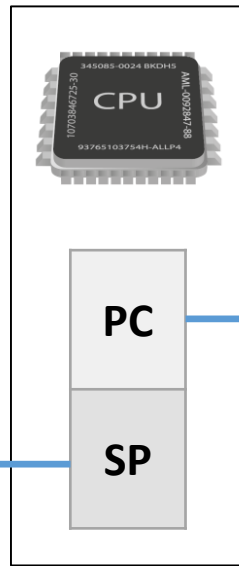
P's kernel stack

```
yield() {
    pid_t old, new;
    old = current_pid();
    new = schedule();
    context_switch(old, new);
}
```

```
context_switch(old, new) {
    int in_progress = 0;
    save_context(old);
    if (in_progress == 1)
        return;
    else
        in_progress = 1;
    restore_context(new);
}
```

OS

## Userspace Process *P*

```
main() {                                    Text
    int i = 10;
    yield();

    i = 20;
    yield();
}
```

```
                                          Stack
i:    10
```

Now yield returns back to user process.
Reset CPU ring to userspace.

**PC**

**SP**

```
                                    context_switch(old, new) {
                                        int in_progress = 0;
                                        save_context(old);
                                        if (in_progress == 1)
                                            return;
                                        else
                                            in_progress = 1;
                                        restore_context(new);
                                    }
```

```
                          yield() {
                              pid_t old, new;
                              old = current_pid();
                              new = schedule();
                              context_switch(old, new);
                          }
---------yield--------
old: P          new: X
:saved SP
:saved PC
```
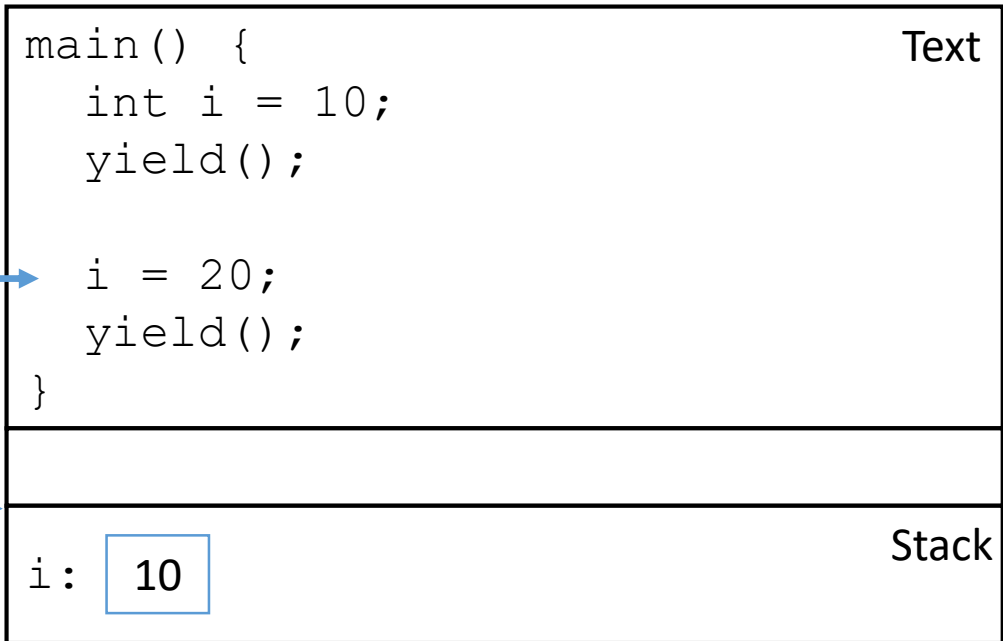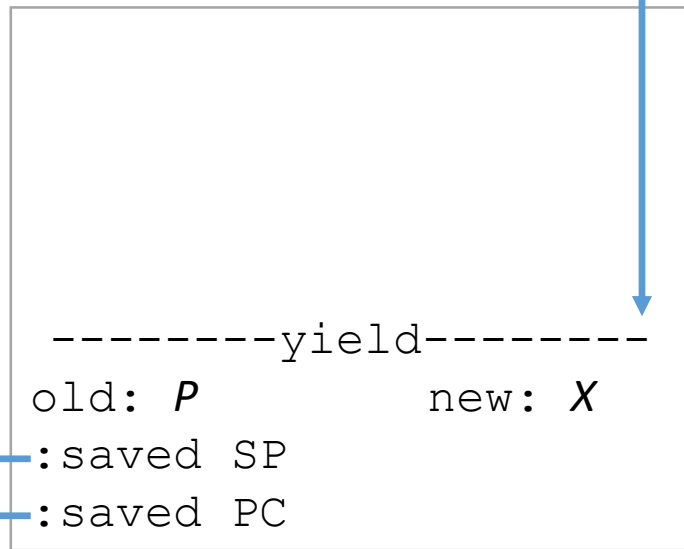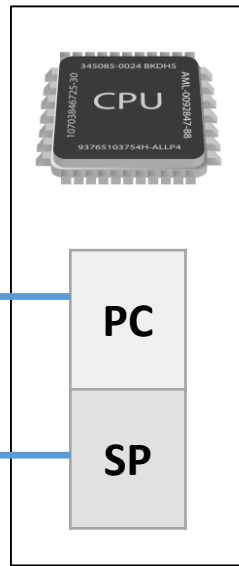
P's kernel stack                          OS

## Userspace Process *P*

```
main() {                           Text
   int i = 10;
   yield();

   i = 20;
   yield();
}
```

```
i:  20                            Stack
```

PC
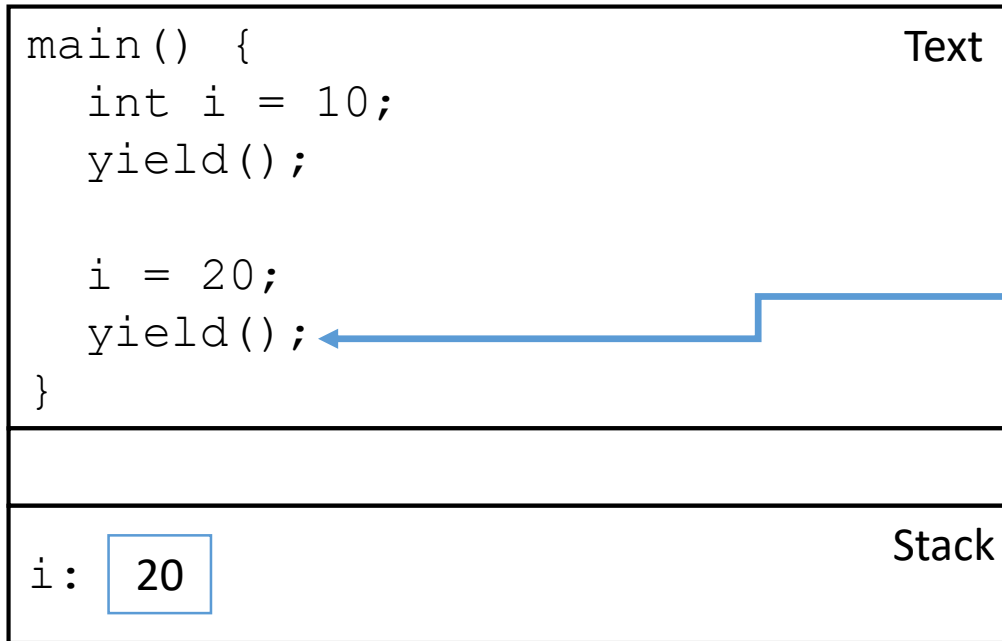
SP

Set i to 20.

---
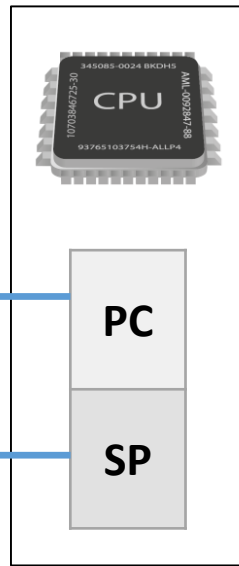
P's kernel stack

```
yield() {
   pid_t old, new;
   old = current_pid();
   new = schedule();
   context_switch(old, new);
}
```

```
context_switch(old, new) {
   int in_progress = 0;
   save_context(old);
   if (in_progress == 1)
      return;
   else
      in_progress = 1;
   restore_context(new);
}
```

OS

## Userspace Process *P*

```
main() {                              Text
    int i = 10;
    yield();

    i = 20;
    yield();
}
```

```
i:   20                               Stack
```

**PC**

**SP**

CPU

Yield again, repeat process.
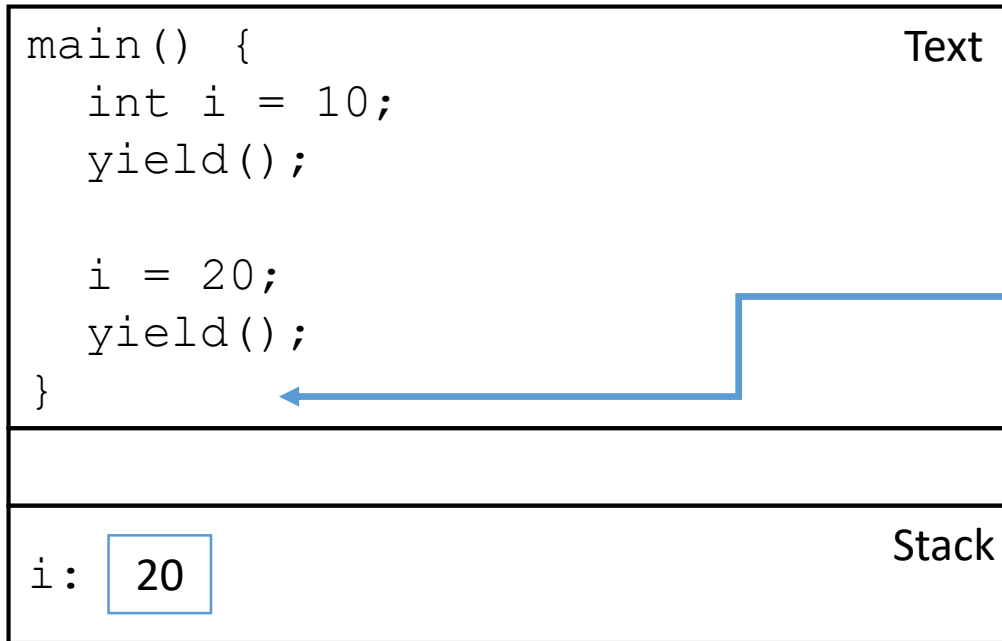
---

P's kernel stack

```
yield() {
    pid_t old, new;
    old = current_pid();
    new = schedule();
    context_switch(old, new);
}
```

```
context_switch(old, new) {
    int in_progress = 0;
    save_context(old);
    if (in_progress == 1)
        return;
    else
        in_progress = 1;
    restore_context(new);
}
```
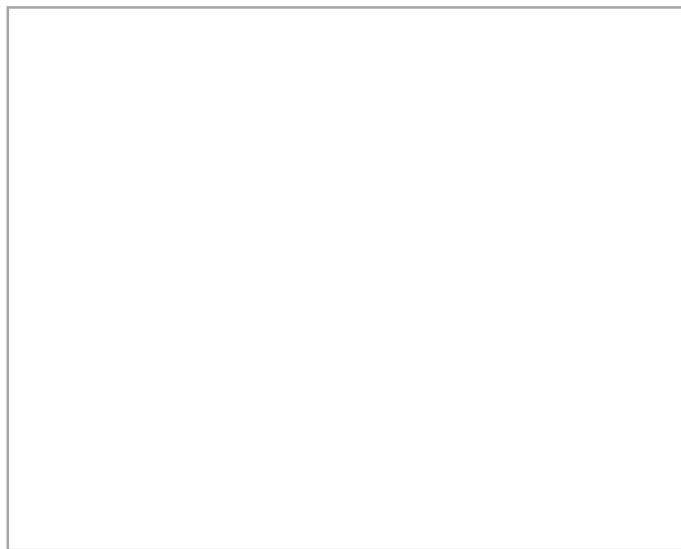
OS

## Userspace Process P

```
main() {                           Text
    int i = 10;
    yield();

    i = 20;
    yield();
}
```

```
                                   Stack
i:  20
```

**CPU**

PC

SP

Eventually, we'll get here again:

Key insight: we just changed a variable on the stack AFTER saving the context!

PCB (P): SP, PC

P's kernel stack

```
  ----context switch----

in_progress:  1

(yield's SP and PC)
  --------yield--------
old: P          new: X
:saved SP
:saved PC
```
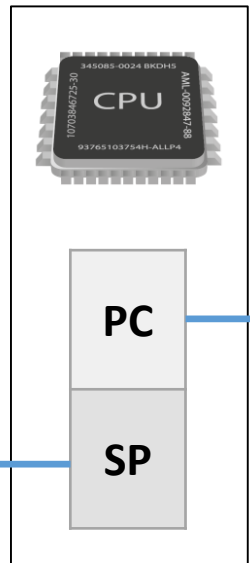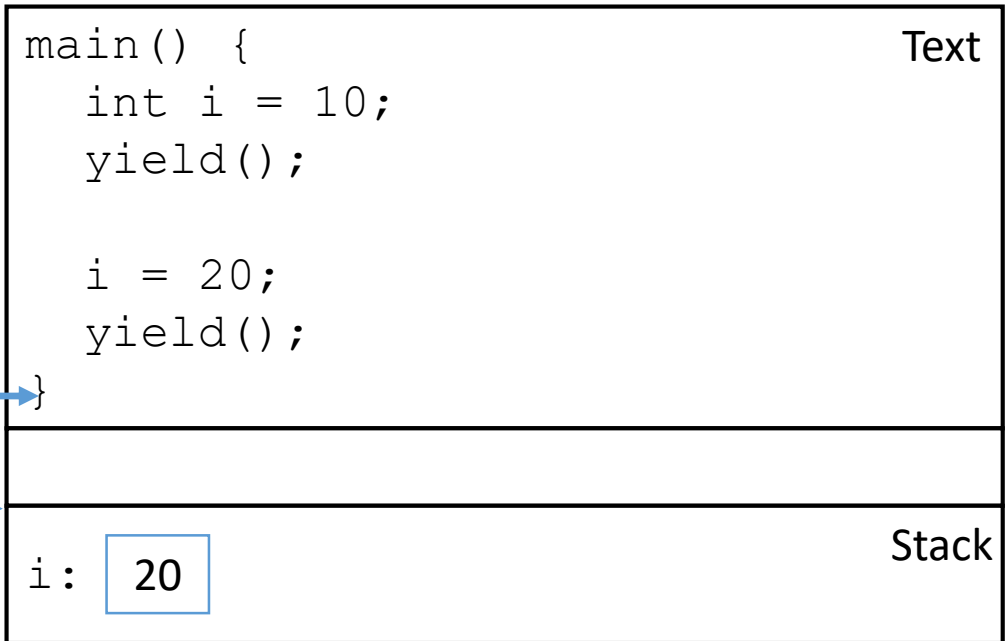
OS

```
yield() {
    pid_t old, new;
    old = current_pid();
    new = schedule();
    context_switch(old, new);
}
```

```
context_switch(old, new) {
    int in_progress = 0;
    save_context(old);
    if (in_progress == 1)
        return;
    else
        in_progress = 1;
    restore_context(new);
}
```
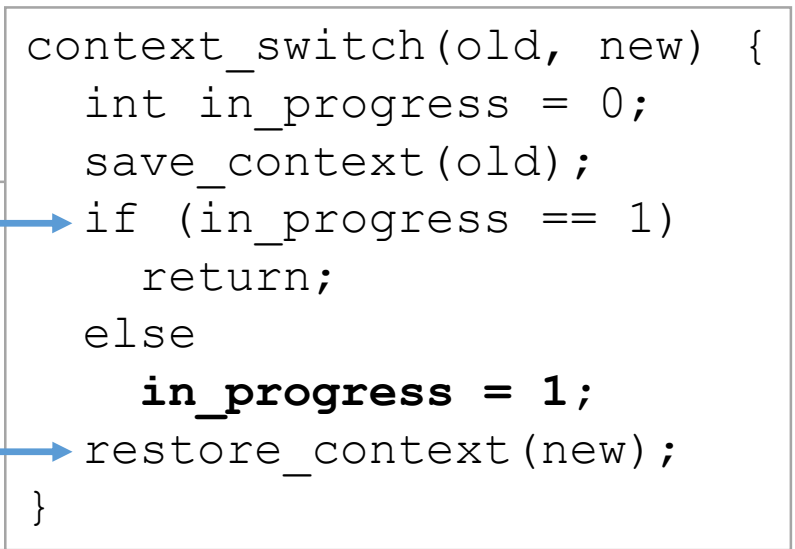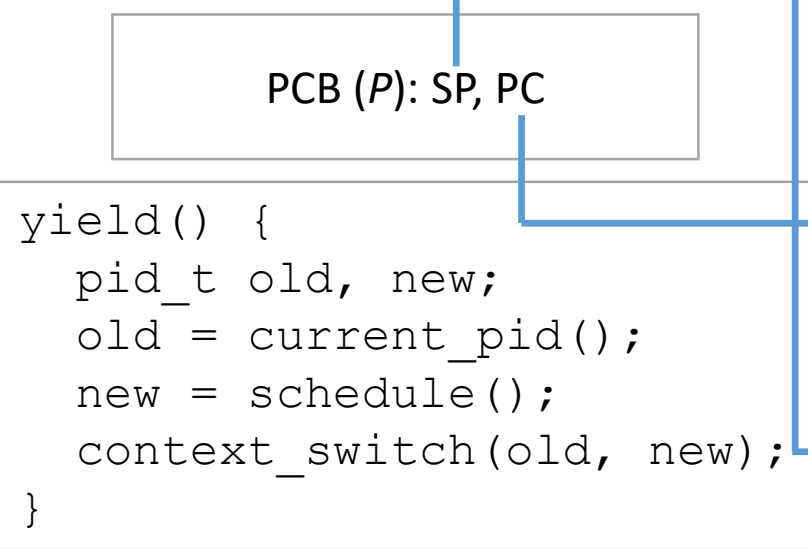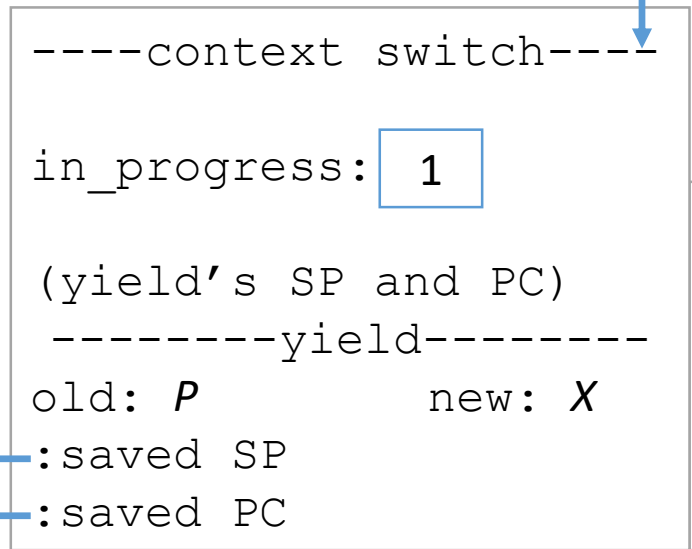
# Context switching questions?

## Userspace Process *P*

```
main() {                                   Text
    int i = 10;
    yield();

    i = 20;
    yield();
}
```

```
                                          Stack

i: [    ]
```

For each call to yield:
How many times does save_context() return?
How many times does restore_context() return?

| Answer | Function | Returns | Function | Returns |
|--------|----------|---------|----------|---------|
| A | save_context | 0 | restore_context | 2 |
| B | save_context | 1 | restore_context | 1 |
| C | save_context | 2 | restore_context | 0 |

```
yield() {
    pid_t old, new;
    old = current_pid();
    new = schedule();
    context_switch(old, new);
}
```
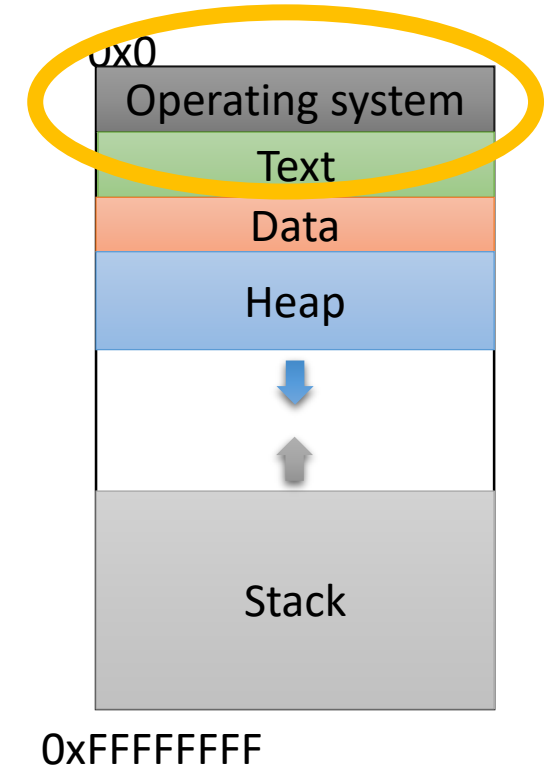
```
context_switch(old, new) {
    int in_progress = 0;
    save_context(old);
    if (in_progress == 1)
        return;
    else
        in_progress = 1;
    restore_context(new);
}
```

P's kernel stack                    OS

# Kernel Execution

- Recall: kernel executes on…
  - process system call
  - process exception
  - hardware interrupt

- Problem: hardware not associated with any process
  - Stack: set aside memory for kernel stack for each <u>CPU</u>

# When should we perform a context switch?

A. When a process makes an I/O system call.

B. When the scheduler decides it's time (how?).

C. Any time the kernel executes (system call, exception, interrupt).

D. Some other time.

# Context Switching: When

- Any time OS executes, check which process to schedule before giving control back to userspace.

- Before giving CPU to a process, set a timer: process's "quantum".

- After quantum expires, timer generates hardware interrupt, giving control back to kernel.
  - Quantum length is controlled by sched policy.

# Summary

- OS stores lots of information about process and resources in PCB.

- OS decides which process to run according to scheduling policy.

- Scheduling is enforced by context switching mechanism.

- Context switch occurs when kernel gets control.
  - Process asked for syscall or caused exception.
  - Hardware interrupt – often the timer device as set by scheduler!