Universität Bamberg

Operating Systems Engineering

Lecture 6: Multitasking

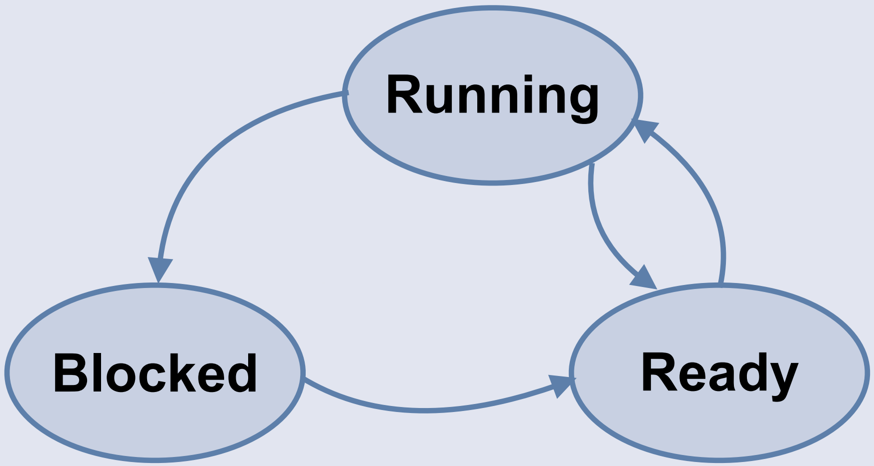
Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insb. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Program vs. process

- So far, we were quite imprecise when using the terms "program" (or "application") and "process"
- We can define both more precisely:
  A ***process*** is a ***program in execution***
- A program is static, represented by the bytes of the executable file on a file system (or in our program header file arrays)
- A process has ***state*** that changes throughout its execution, e.g.:
  - The current program counter (instruction pointer)
  - The values of processor registers
  - The values of variables
  - In addition, information about resources, user IDs, permissions…
- All this state has to be **saved** when the process is not active (running)

Process behavior

- In our current implementation, a process can either be **running** (it currently uses the processor) or **ready** (it has to wait until it is scheduled)
- In a more general process model, a process can also be **blocked**, i.e. waiting for the completion of an I/O operation it requested
- Each process transitions between these states:



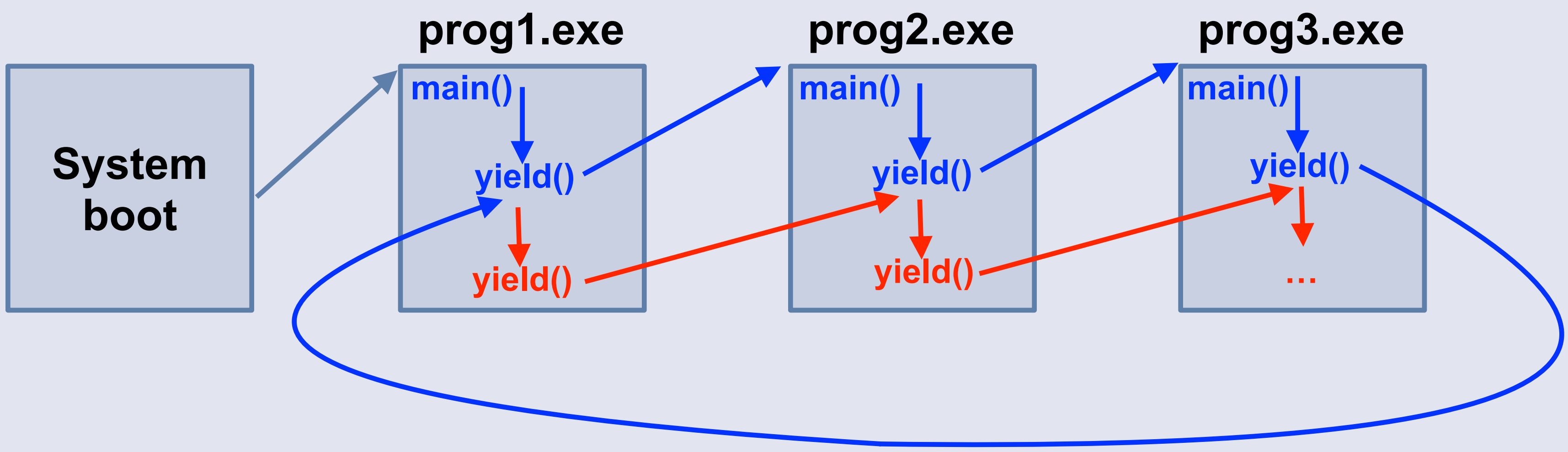
Questions:

- How many processes can be in state "Running" at the same time on a single processor system?
- Why is there no transition from "Blocked" to "Running"?
- At which points in time do the transitions occur?
- What happens if all processes are blocked?

Cooperative Multitasking

Universität Bamberg

- Cooperative multitasking uses a `yield()` system call to switch between processes ***without terminating the calling process***
    - This requires more saving of state

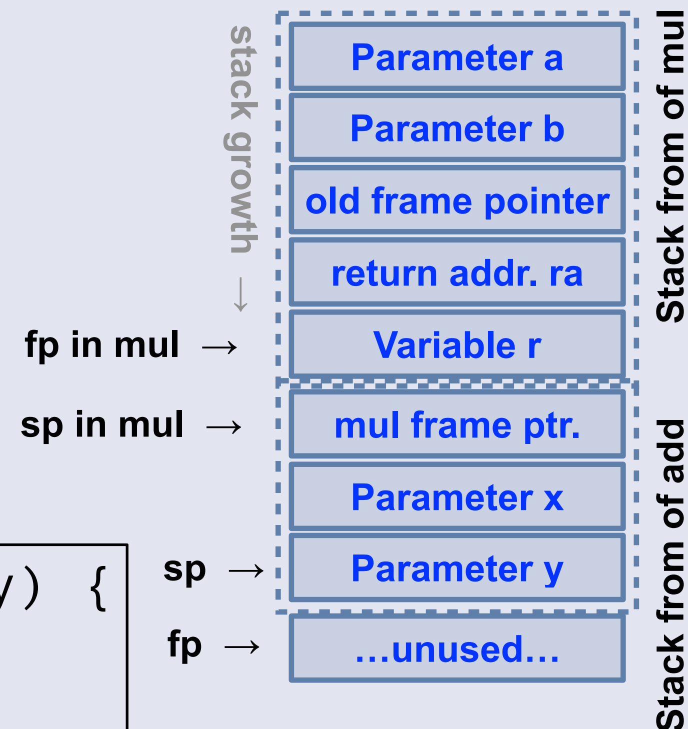


- A program calling `yield()` continues to execute after the call to yield
    - (almost) like a regular system call…

Cooperative multitasking and the stack

Universität Bamberg

- The stack is used (by code generated by the compiler) to store the following *process state*:
  - local variables for each function in the call hierarchy
  - return addresses to a function higher up in the call hierarchy
- Example: function mul calls add
  - Add is a leaf function (calls no other function), does not require saving of the return address register `ra`



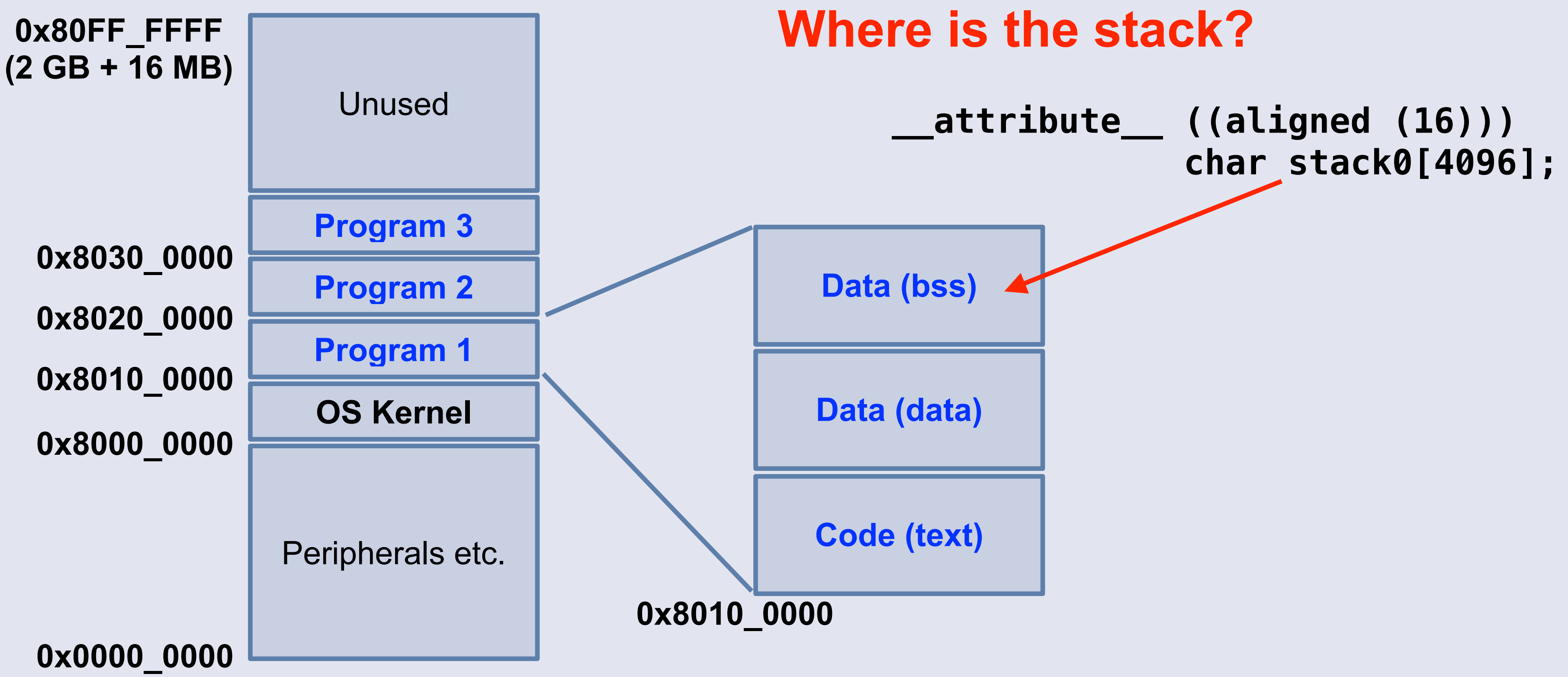

```
int mul(int a,b) {
    int r = 0;
    while (a--)
      r = add(r,b);
    return r;
}
```

```
int add(int x,y) {
  return x+y;
}
```

Cooperative multitasking: memory view

Universität Bamberg

- **All processes have to remain in memory now**
  - Code (text segments) is static and could be reloaded, but the data programs manipulate (data + bss segments) would need to be saved



Unfortunate effects…

Universität Bamberg

- In our current setup, this still requires a separate linker script for each of the user processes

- **Idea:** parametrize the linker to dynamically define a start address for each of the processes (to 0x8000_0000 + 0x0010_0000 * process ID) using a linker option on the gcc command line:

  ```
  -Wl,--defsym=PROC_START=$(VALUE_TO_OVERRIDE)
  ```

  (see [4] for details)

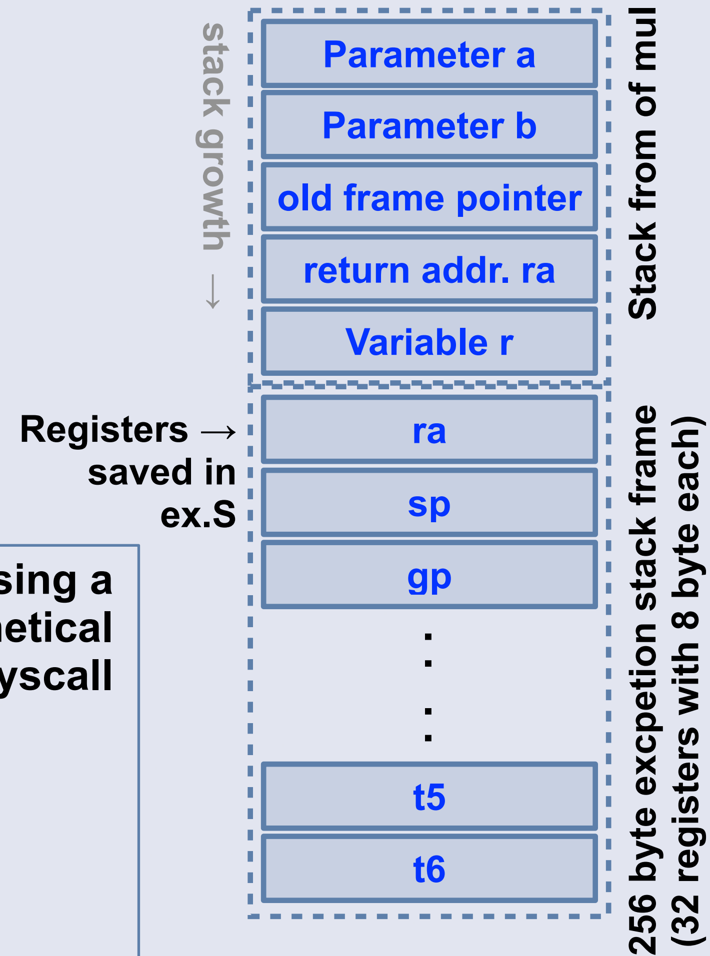
- Virtual memory will make our life easier (later…)
  - All programs can be linked at the same address
  - The virtual memory subsystem uses the MMU to map the overlapping virtual address ranges of processes to separate virtual memory regions

What other data is important?

- *Register values* – we only have one set of registers for all processes
  - Executing another process (or the kernel code) overwrites values!
- We already have ex.S doing this
  - This already saves all registers to the process stack – even those saved by the `exception` C function prologue code

```
int mul(int a,b) {
    int r = 0;
    asm("mv a0, %0"::"r" (a));
    asm("mv a1, %0"::"r" (b));
    asm("li a7, 99");
    asm("ecall");
    asm("mv %0, a0":"=r" (r));
    return r;
}
```

mul using a hypothetical mul syscall



A stack for the OS

Universität Bamberg

- Where does the OS store its own context?
  - In its own address range (0x8000_0000–0x800F_FFFF)
- Currently, there is not a lot of context to be saved:
  - `exception` only relies on
    - the `current_process` process ID (global variable in the data segment)
    - the `pcb` process control block array of `struct pcb_struct` (one entry per process), also in data
- What if we call other functions from `exception` that could return to user mode before they are finished, blocking I/O functions?
  - We would need a stack to store register values and the call hierarchy
    - This information could be stored on the current user process stack
    - Disadvantage: could be manipulated by a user process?
- **Idea:** provide a stack for the kernel and switch to/from it when entering/leaving M mode

Scheduling

Universität Bamberg

- Simple implementation of a **round-robin scheduler:** the OS switches to a different process at every invocation of `yield()` – here shown for two processes
- The `pcb` struct holds the current **state** of the process
  - At the moment, we need the PC and SP at the point the exception was invoked (i.e. the `ecall` instruction in user mode)

Simple scheduler saving process context in exception (kernel.c)

```
case 23: // yield system call
         // save pc, stack pointer
         pcb[current_process].pc = pc;
         pcb[current_process].sp = s;

         // select new process
         current_process++;
         if (current_process > 2) current_process = 0;
         pc = pcb[current_process].pc; // add +4 later!

         break;
```

Process state

Universität Bamberg

- Where can we get the values of the user process program counter `pc` and stack pointer from?
- The value of `pc` (address of the `ecall` instruction) was stored by the processor in when the exception was invoked
- The value of `s` needs to come from the assembler code in `ex.S`
  - Passed as parameter to our `exception` function
  - We can now use the values of a0 and a7 (parameter and system call number) stored on the stack frame in ex.S:

Read syscall nr, parameter and exception PC value

```
void exception(stackframe *s) {
  uint64 pc, nr, param;

  nr    = s->a7;   // read syscall number from stack
  param = s->a0;   // read parameter from stack

  pc = r_mepc();   // read exception PC
```

Passing the stack pointer

- The value of the user stack pointer `s` needs to come from the assembler code in `ex.S`
  - Passed as parameter to our `exception` function (in a0)
  - This is the stack pointer value after we saved all registers
- Now `a0` no longer contains the system call parameter!
  - Take this (and `a7`) from the user process stack as shown on the previous slide!

Stack handling in ex.S

```
ex:
   addi sp, sp, -256
   sd ra, 0(sp)
   …
   sd t6, 240(sp)

   // set stackframe parameter
   mv a0, sp

   // call the C trap handler
   call exception

   // restore stackframe
   mv sp, a1

   // restore registers.
   ld ra, 0(sp)
   // ld sp, 8(sp)
   ld gp, 16(sp)
   …
   ld t6, 240(sp)
   addi sp, sp, 256
   mret
```

Protection, again

Universität Bamberg

- Idea: Enable additional protection
  - So far: protect kernel from program accesses
  - Additional: **protect programs against each other**
- We now have to define PMP regions as follows in `setup.c`:
  - 0x0000_0000–0x7FFF_FFFF: I/O
  - 0x8000_0000–0x800F_FFFF: kernel
  - 0x80**n**0_0000–0x80**n**F_FFFF: process **n**

Switching protection in the exception handler (kernel.c)

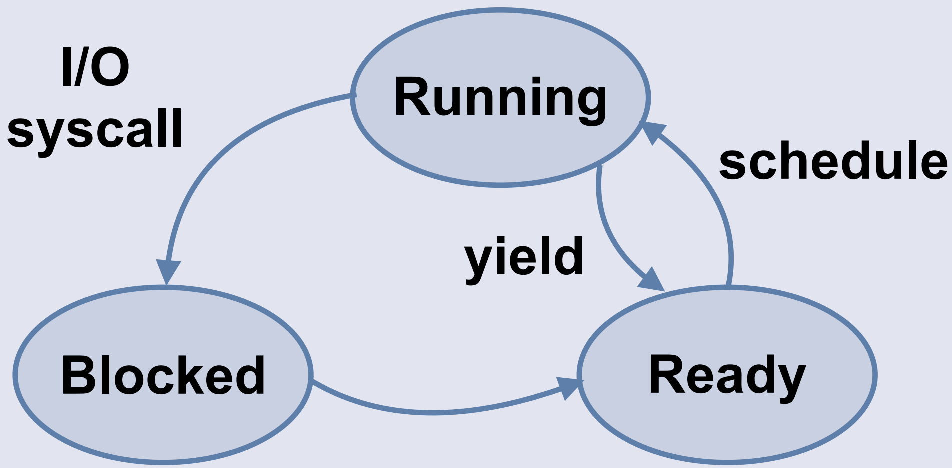
```
case 23: // yield system call
  // save pc, stack
  …
  // Switch memory protection to new process
  if (current_process == 0)
    w_pmpcfg0(0x0f0000);  // 2 - full access; 1,0 - no access
  else if (current_process == 1)
    w_pmpcfg0(0x0f000000); // 3 - full; 2,1,0 - no access
  // …handle further processes in a more elegant way?
  break;
```

Cooperation extended

- Idea: Also enable switching at every system call
- This will enable us to implement our three-state process model
  - Running, Ready, Blocked
  - Switch from Running to Blocked when an I/O syscall is made
- This also enables *event loops* for interactive applications (see next slide)

Question:

- What happens in our process state diagram if a process calls syscall #42 (exit)?



Cooperative MT: Event loops

Universität Bamberg

- Event loops allow the OS to take back control ***often***
  - …but not always
  - relies on programmer discipline (and bug-free programs…)

Example event loop application code for Mac System 7

```
PROCEDURE MyEventLoop;
VAR
   cursorRgn:     RgnHandle;
   gotEvent:      Boolean;
   event:         EventRecord;
BEGIN
   cursorRgn := NewRgn; {pass an empty region the first time thru}
   REPEAT
      gotEvent := WaitNextEvent(everyEvent, event, MyGetSleep,
                                         cursorRgn);
      IF (event.what <> kHighLevelEvent) AND (NOT gInBackground)
         THEN MyAdjustCursor(event.where, cursorRgn);
      IF gotEvent THEN  {the event isn't a null event, }
         DoEvent(event) { so handle it}
      ELSE              {no event (other than null) to handle }
         DoIdle(event); { right now, so do idle processing}
   UNTIL gDone;         {loop until user quits}
END;
```

References

1. Andrew Waterman, Krste Asanovic and John Hauser, *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Document,* Version 20211203
2. *SiFive Interrupt Cookbook*, https://starfivetech.com/uploads/sifive-interrupt-cookbook-v1p2.pdf
3. Free Software Foundation, *Debugging with gdb*, 10th Edition, section 10.19 https://sourceware.org/gdb/onlinedocs/gdb/Dump_002fRestore-Files.html
4. StackOverflow, *GNU LD: How to override a symbol value*, https://stackoverflow.com/questions/10032598/gnu-ld-how-to-override-a-symbol-value-an-address-defined-by-the-linker-script