# Operating Systems (234123)
# *Processes & Signals*

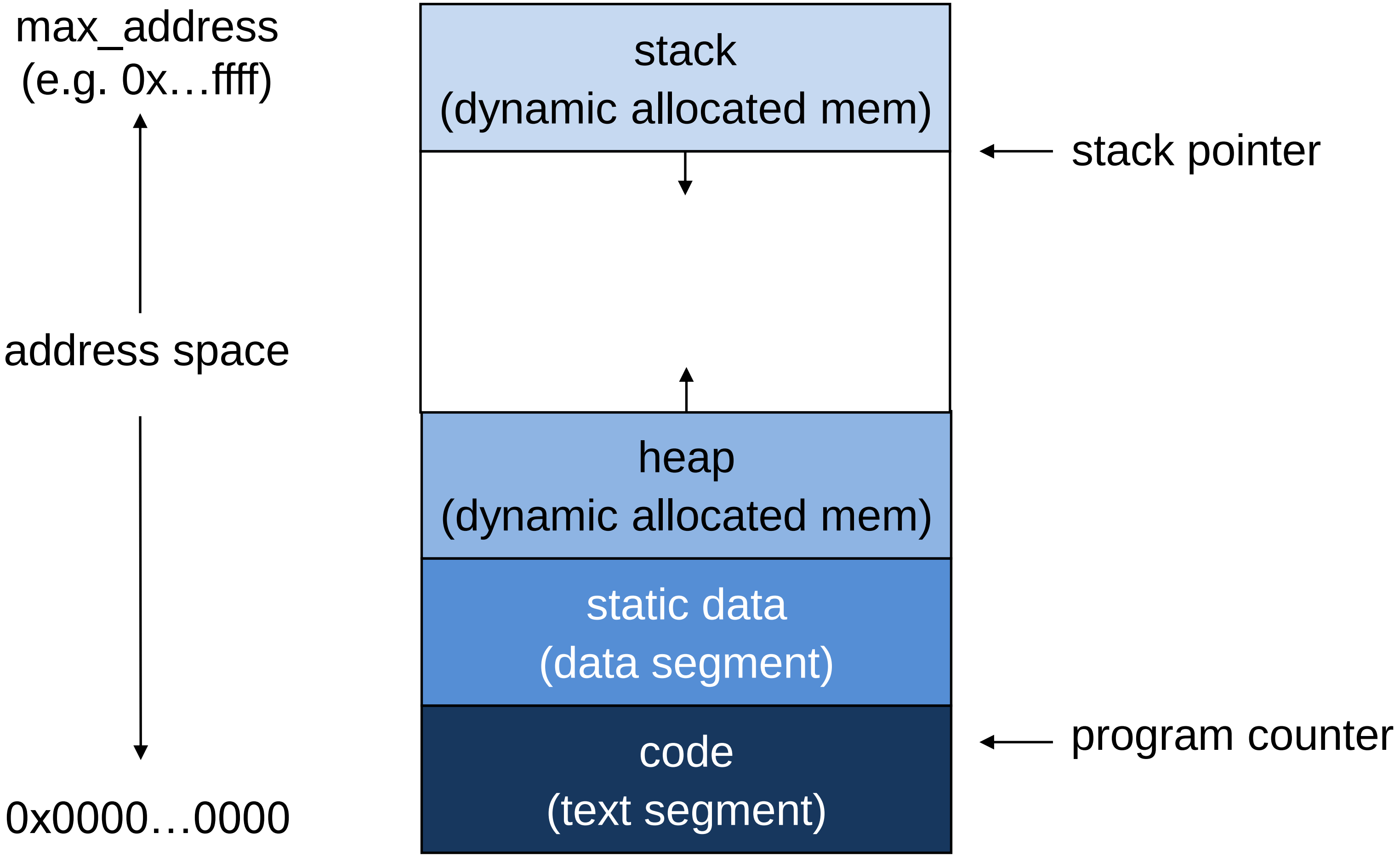
Dan Tsafrir

2024-06-03, 2024-06-10

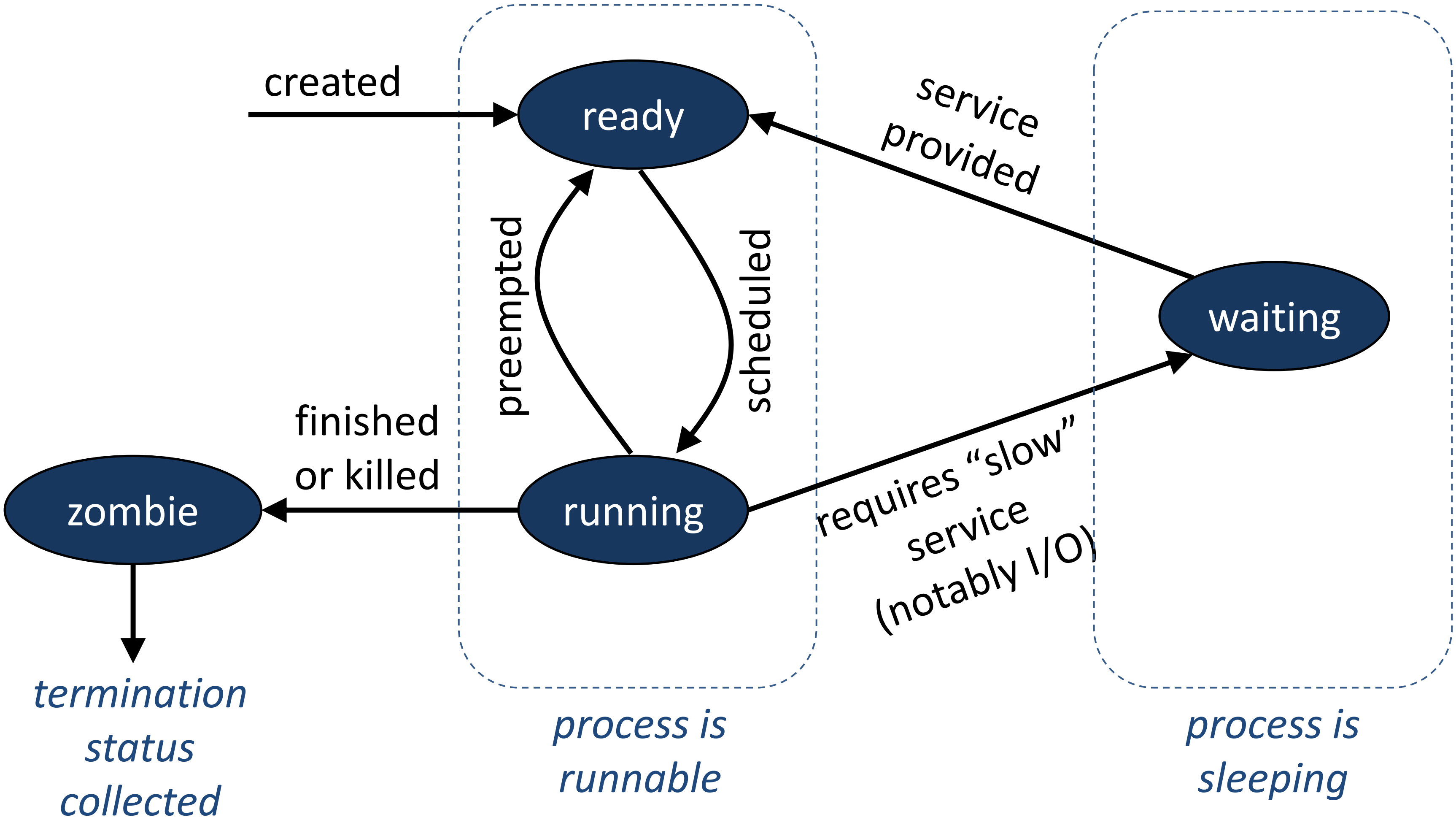
# What's a process

- **An implementation of the abstract machine concept**
  - Which we discussed in the previous lecture
- **A running instance of an executable, invoked by a user**
  - Can have multiple independent processes of the same executable
- **A schedulable entity, on the CPU**
  - OS decides which of these entities gets to run on a CPU core, and when
- **Sometimes called**
  - Task or job
- **The OS kernel is neither a process nor a schedulable entity**
  - Rather, it's a set of procedures executing in response to events (≈ interrupts)
  - Albeit sometimes the OS runs some code within schedulable entities
    - But then we prefer not to refer to these entities as "processes", which correspond to *user* programs; we may refer to them as "kernel threads" instead

# Process address space is contiguous



# Process states



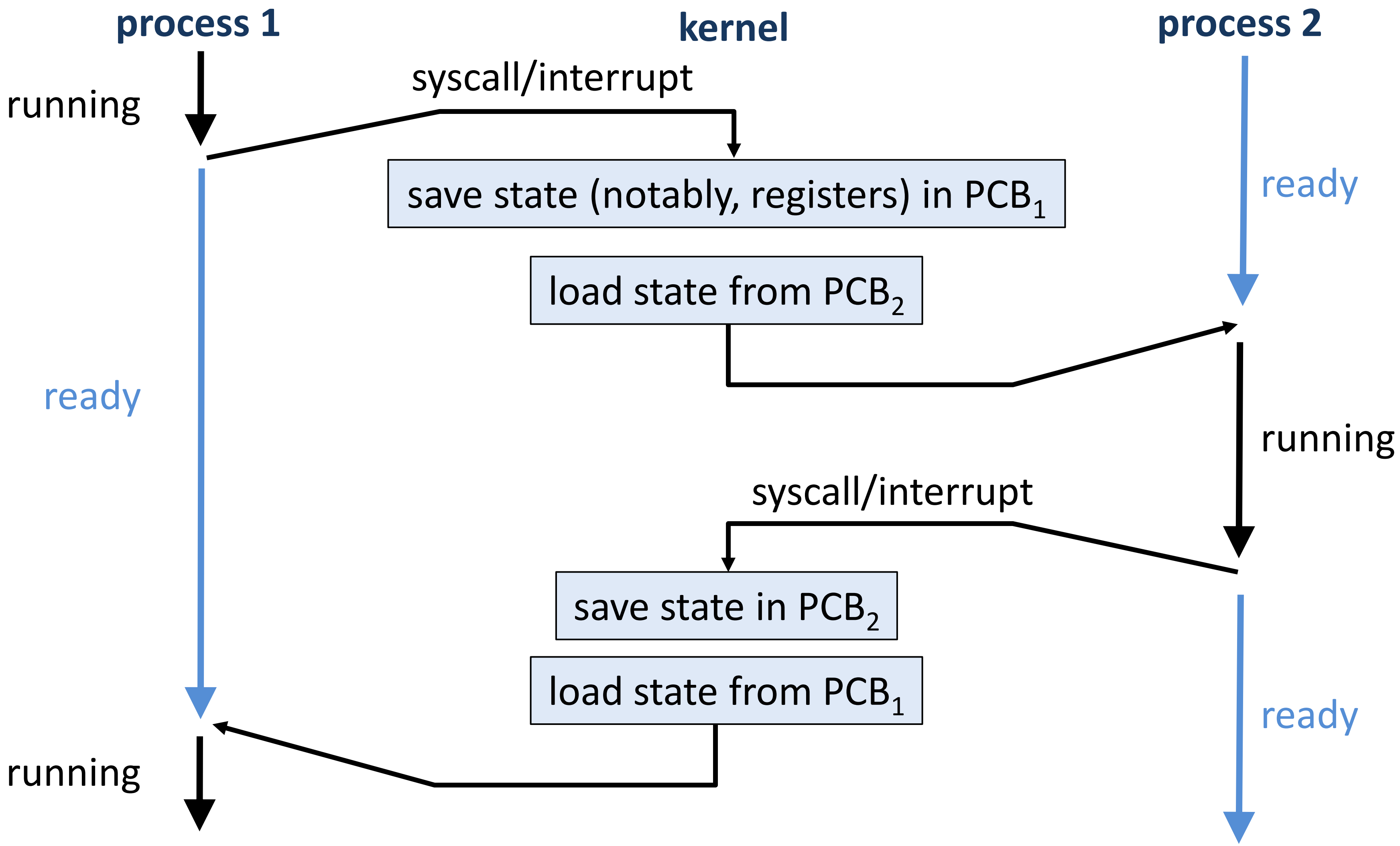
# Process control block ("PCB")

- **The OS maintains a "state" for every process**
  - Encapsulated in a  PCB
- **In Linux**
  - Called a "process descriptor"
  - Of type task_struct (C struct)
  - Has O(100) fields
- **Used in context switches**
  - Updated upon preemption
  - Loaded upon resumption
- **Question**
  - Can a process access its own PCB?

- **PID (process ID)**
- **UID (user ID)**
- **Pointer to address space**
- **Registers**
- **Scheduling priority**
- **Resources usage limits (e.g., memory, CPU, num of open files)**
- **Resources consumed**
- **State (previous slide)**
- **Current/present working directory (pwd=cwd)**
- **Open files table**
- **…**

process attributes in PCB

# Context switching (in runnable state)



process 1

kernel

process 2

running

syscall/interrupt

save state (notably, registers) in $PCB_1$

ready

load state from $PCB_2$

ready

running

syscall/interrupt

save state in $PCB_2$

load state from $PCB_1$

ready

running

# Process creation & termination

- **One process (the “parent”) can create another (the “child”)**
  - A new PCB is allocated and initialized
  - Homework: run ‘ps auxwww’ in the shell; PPID is the parent’s PID
- **In POSIX, child process inherits most of parent’s attributes**
  - UID, open files (should be closed if unneeded; why?), cwd, etc.
- **While executing, PCB moves between different queues**
  - According to state change graph
  - Queues: runnable, sleep/wait for event i (i=1,2,3…)
- **After a process dies (exit()s / interrupted), it becomes a zombie**
  - Parent uses wait* syscall to clear zombie from the system (why?)
  - Wait syscall family: wait, waitpid, waitid, wait3, wait4; example:
  - pid_t wait4(pid_t, int *wstatus, int options, struct rusage *rusage);
- **Parent can sleep/wait for its child to finish or run in parallel**
  - wait*() will block unless WNOHANG given in ‘options’
  - Homework: read ‘man 2 wait’

# fork() – spawn a child process

- **fork() initializes a new PCB**
  - Based on parent’s value
  - PCB added to runnable queue
- **Now there are 2 processes**
  - At same execution point
- **Child’s new address space**
  - Complete copy of parent’s space, with one difference…
- **fork() returns twice**
  - At the parent, with pid>0
  - At the child, with pid=0
- **What’s the printing order?**
- **‘errno’ – a global variable**
  - Holds error num of last syscall

```
int main(int argc, char *argv[])
{
  int pid = fork();
  if( pid==0 ) {
      //
      // child
      //
      printf(“parent=%d son=%d\n”,
             getppid(), getpid());
  }
  else if( pid > 0 ) {
      //
      // parent
      //
      printf(“parent=%d son=%d\n”,
             getpid(), pid);
  }
  else { // print string associated
         // with errno
      perror(“fork() failed”);
  }
  return 0;
}
```

# System call errors

```
// int errno = number of last system call error.
// Errors aren’t zero. (If you want to test value of
// errno after a system call, need to zero it before.)
#include <errno.h> // see man 3 errno

// const char * const sys_errlist[];
// char* strerror(int errnum) {
//    // check errnum is in range
//    return sys_errlist[errnum];
// }
#include <string.h>

//  void perror(const char *prefix);
// prints: “%s: %s\n” , prefix, sys_errlist[errno]
#include <stdio.h>
```

# exec*() – replace current process image

- **To start an entirely new program**
  - Use the exec*() syscall family; for example:
    - int execv(const char *progamPath, char *const argv[]);
  - Homework: read ‘man execv’
- **Semantics**
  - Stops the execution of the invoking process
  - Loads the executable ‘programPath’
  - Starts ‘programPath’, with ‘argv’ as its argv
  - Never returns (unless fails)
  - *Replaces* the new process; doesn’t create a new process
    - In particular, PID and PPID are the same before/after exec*()